The VLDB Journal (2021) 30:641–665
https://doi.org/10.1007/s00778-021-00663-8

REGULAR PAPER

Micro-architectural analysis of in-memory OLTP: Revisited

Utku Sirin[1] · Pınar Tözün[2] · Danica Porobic[3] · Ahmad Yasin[4] · Anastasia Ailamaki[1]

Received: 15 May 2020 / Revised: 11 January 2021 / Accepted: 17 March 2021 / Published online: 31 March 2021
© The Author(s) 2021

Abstract
Micro-architectural behavior of traditional disk-based online transaction processing (OLTP) systems has been investigated extensively over the past couple of decades. Results show that traditional OLTP systems mostly under-utilize the available micro-architectural resources. In-memory OLTP systems, on the other hand, process all the data in main-memory and, therefore, can omit the buffer pool. Furthermore, they usually adopt more lightweight concurrency control mechanisms, cache-conscious data structures, and cleaner codebases since they are usually designed from scratch. Hence, we expect significant differences in micro-architectural behavior when running OLTP on platforms optimized for in-memory processing as opposed to disk-based database systems. In particular, we expect that in-memory systems exploit micro-architectural features such as instruction and data caches significantly better than disk-based systems. This paper sheds light on the micro-architectural behavior of in-memory database systems by analyzing and contrasting it to the behavior of disk-based systems when running OLTP workloads. The results show that, despite all the design changes, in-memory OLTP exhibits very similar micro-architectural behavior to disk-based OLTP: more than half of the execution time goes to memory stalls where instruction cache misses or the long-latency data misses from the last-level cache (LLC) are the dominant factors in the overall execution time. Even though ground-up designed in-memory systems can eliminate the instruction cache misses, the reduction in instruction stalls amplifies the impact of LLC data misses. As a result, only 30% of the CPU cycles are used to retire instructions, and 70% of the CPU cycles are wasted to stalls for both traditional disk-based and new generation in-memory OLTP.

Keywords OLTP · Workload characterization · In-memory OLTP systems · Micro-architectural analysis

1 Introduction

Recent years have witnessed the rise of in-memory or main-memory optimized OLTP systems [10,33,56]. Traditional OLTP engines are disk-based since they are designed in an era where the main-memory size was in megabytes. Today, however, a server hardware with 1TB main-memory is a commodity. Therefore, the database management systems (DBMS) are able to process the data working set of most OLTP applications in-memory. This has led various vendors and researchers to design brand-new OLTP engines optimized for the case where the dataset resides in memory [23,26,27,52].

In-memory OLTP systems have significant differences compared to disk-based systems. First, since the data working set resides mostly in memory, in-memory OLTP systems omit the buffer pool component, which acts as the virtual memory of a DBMS and is, therefore, essential for the disk-based systems. Then, they tend to adopt more lightweight

Danica Porobic: The work was done while the author was at EPFL.

✉ Utku Sirin
utku.sirin@epfl.ch

Pınar Tözün
pito@itu.dk

Danica Porobic
danica.porobic@oracle.com

Ahmad Yasin
ahmad.yasin@intel.com

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch

1 EPFL, Lausanne, Switzerland

2 IT University of Copenhagen, Copenhagen, Denmark

3 Oracle, Redwood City, USA

4 Intel Corporation, Mountain View, USA

concurrency control mechanisms to avoid the scalability bottlenecks that arise due to traditional centralized locking. They also opt for cache-conscious indexes instead of the disk-optimized B-trees. Finally, since their codebases are written from scratch, they tend to have lighter storage engines in terms of the instruction footprint.

OLTP benchmarks are famous for their suboptimal micro-architectural behavior. There is a large body of work that characterizes OLTP benchmarks at the micro-architectural level [5,12,22,41,50,53,54]. They all conclude that OLTP exhibits high stall time ($>$ 50% of the execution cycles) and a low instructions-per-cycle (IPC) value ($<$ 1 IPC on machines that can retire up to 4 instructions in a cycle) [12]. The instruction cache misses that mainly stem from the large instruction footprint of transactions are the main source of the stall time, while the next contributing factor is the long-latency data misses from the last-level cache (LLC) [53].

All the previous workload characterization studies, however, run the OLTP benchmarks on a disk-based OLTP engine. Considering the lighter components, cache-friendly data structures, and cleaner codebase of in-memory systems, one expects them to exhibit better cache locality (especially for the instruction cache) and less memory stall time. Due to the distinctive design features of the in-memory systems from the disk-based ones, however, it is not straightforward to extrapolate how OLTP benchmarks behave at the micro-architectural level when run on an in-memory engine solely by looking at the results of previous studies.

In this paper, we perform a detailed analysis of the micro-architectural behavior of the in-memory OLTP systems. More specifically, we compare three in-memory OLTP systems (an in-memory OLTP engine of a popular commercial vendor, a ground-up designed in-memory OLTP system, and an open-source OLTP engine, Silo [55]) to two disk-based OLTP systems (a popular commercial DBMS and an open-source OLTP engine, *Shore-MT* [43]). We examine CPU cycles breakdowns while running simple micro-benchmarks as well as the more complex TPC benchmarks (TPC-B and TPC-C) [1]. Our analysis demonstrates the following:

- Despite all the design differences, in-memory OLTP spends more than half of the execution cycles in memory stalls, either due to instruction or data cache misses, similar to disk-based OLTP.
- Popular commercial OLTP systems that rely on legacy codebases mainly suffer from instruction cache misses due to their large and complex instruction footprint, regardless the system is disk-based or in-memory. Even though the in-memory optimized DBMS components reduce the total instruction footprint at the storage manager side, the instruction footprint and code complexity of the rest of the components overshadow the benefits of these optimizations at the micro-architectural level.
- Ground-up designed in-memory OLTP systems do not suffer from instruction cache misses thanks to their relatively shorter and simpler instruction footprint compared to the popular commercial systems relying on legacy codebases. However, ground-up designed in-memory OLTP systems spend more than half of their execution time in data cache miss stalls due to the random data accesses the OLTP workloads do. As a result, their CPU utilization characteristics remain close to the popular commercial OLTP systems relying on legacy codebases.

This paper revisits [48] by using the methodology proposed by [49] on a later generation Intel processor. The newly used methodology allows end-to-end execution time profiling rather than merely relying on cache miss counts. The new generation of Intel processor reveals a significant change in the micro-architectural behavior of the OLTP systems. In particular, the main performance bottleneck of the disk-based and ground-up designed in-memory OLTP system shifts from instruction cache miss stalls to data cache miss stalls thanks to the improvements in the core micro-architecture. We update the contributions and the text according to the newly seen micro-architectural behavior. Furthermore, we compare and contrast Intel's two successive processor generations and reason about the significant change in the micro-architectural behavior. We use *Silo* kernel OLTP engine instead of *HyPer*, as *HyPer* is not publicly available anymore. Using *Silo* allows assessing the complexity of the core of an in-memory OLTP system and hence making the difference between an end-to-end OLTP system and an OLTP kernel engine. Finally, we analyze memory bandwidth and commonly available acceleration features—hyper-threading, turbo-boost, and hardware prefetchers—completing the hardware–software interaction analysis of OLTP on modern processors.

The rest of the paper is organized as follows. Section 2 gives an overview of the in-memory OLTP systems-and surveys-related work on workload characterization and micro-architectural analysis studies. Section 3 describes the experimental methodology. Sections 4 and 5 present the analysis results with a micro-benchmark and TPC benchmarks, respectively. Section 6 analyzes the effects of transaction compilation, index structures, and data types, whereas Sect. 7 investigates the impact of multi-threading on the micro-architectural behavior. Section 8 presents the analysis of the memory bandwidth consumptions. Section 9 analyzes the acceleration features such as hyper-threading, turbo-boost, and hardware prefetchers. Section 10 compares two latest generations of Intel's successive micro-architectures. Finally, Sect. 11 discusses the results and Sect. 12 concludes.

2 Background and related work

In-memory DBMS gained a lot of popularity in the last decade. In Sect. 2.1, we detail the underlying factors for this trend and the main design characteristics of in-memory OLTP systems. Then, in Sect. 2.2, we go over the recent workload characterization studies that focus on OLTP applications and highlight why they are not representative for in-memory OLTP systems.

2.1 In-memory OLTP

Commodity servers of the last decade follow two fundamental trends: (1) main-memory becoming cheaper and (2) number of cores increasing exponentially. Simply increasing buffer pool size and number of worker threads to exploit the large main-memory and all the available cores, respectively, lead to marginal gains. Therefore, these two hardware trends have triggered alternative architectures for new-generation DBMS.

As DRAM prices become cheap enough to buy 1TB main-memory for ∼$30 K, today it is possible for most OLTP applications to keep all of their data working set in main-memory. This has led to the development of various in-memory or main-memory optimized OLTP systems. These systems either manage all the data in main-memory or make sure that the hot data resides in main-memory. Since they manage to eliminate/minimize the disk I/O for the data page accesses, the overheads associated with managing the buffer pool overweigh its benefits [16]. Therefore, the in-memory OLTP systems omit the buffer pool component even though it is essential for the traditional disk-based DBMS.

On the other hand, in step with Moore's law, the hardware vendors keep providing more and more opportunities for parallelism. Modern servers tend to have multiple multi-core processors in the same machine and allow OLTP systems to handle increasing number of transactional requests in parallel. However, the traditional concurrency control mechanisms using a centralized lock manager and two-phase locking are designed at an era where the server hardware were uniprocessors. Therefore, they do not scale on multi-cores preventing OLTP systems from exploiting the sheer number of cores available to them [37,61].

In order to achieve better scalability, in-memory OLTP systems adopt alternative concurrency control mechanisms. These mechanisms can be broadly grouped into two categories based on whether they partition the data or not. The ones that partition the data use one data partition for each core and a single worker thread for each partition. Systems like VoltDB [52] (or its ancestor H-Store [51]) and the initial version of HyPer [23] deploy this approach. As a result, they avoid any form of locking within a partition and need to coordinate worker threads only when a transaction is multi-partition. The systems that prefer avoiding any kind of data partitioning, like Hekaton [26], SAP HANA [27], or the latest version of HyPer [36], rely on optimistic and multi-version concurrency control [7].

In addition to alternative concurrency control mechanisms, in-memory database systems also deploy cache-conscious index structures. They align the index page sizes to the size of a cache line as opposed to the size of a disk page and/or adopt lock-free index page access mechanisms rather than using traditional page latches [28,55]. Moreover, the in-memory OLTP systems tend to depend on pre-determined stored procedures instead of ad hoc queries [23,26,52] and apply efficient compilation optimization techniques that optimize the instruction stream for a particular transaction [26,35]. Finally, the new-age in-memory OLTP systems have codebases that are implemented from scratch. Therefore, they are expected to have a cleaner codebase compared to the traditional disk-based systems where the codebase consists of many branch statements and obsolete code paths due to different release versions spanning several decades of development.

Overall, in-memory OLTP engines deploy lighter storage manager components compared to the traditional disk-based systems aiming to utilize the resources of the modern server hardware in a more effective way.

2.2 OLTP at the micro-architectural Level

There is a large body of related work analyzing the micro-architectural behavior of OLTP workloads. Barrosso et al. [5] investigate the memory system behavior of OLTP and DSS style workloads both on a real machine and with a full-system simulation. They argue that these two types of workloads would benefit from different architectural designs in terms of the memory system. Ranganathan et al. [41] perform a similar analysis. However, they only focus on the effectiveness of out-of-order execution on SMPs while running these workloads in a simulation environment. On the other hand, Keeton et al. [22] and Stets et al. [50] experiment only with OLTP benchmarks (TPC-B and TPC-C) on real hardware. All of these studies agree that OLTP workloads utilize the underlying micro-architectural resources very poorly.

Ailamaki et al. [2] examine where the time goes on four commercial DBMS using a micro-benchmark to have a fine-grain understanding of the memory system behavior on multi-processors, whereas Hardavellas et al. [15] analyze TPC-C and TPC-H on both in-order and out-of-order machines in a simulation environment. These studies focus on the implications for the DBMS rather than the hardware to achieve better hardware utilization.

More recent workload characterization studies [12,54] additionally analyze the TPC-E benchmark and show that micro-architecturally TPC-E behaves very similarly to the

TPC-B and TPC-C benchmarks. These studies also corroborate the findings of the previous studies in terms of the inefficient use of the memory hierarchy when running OLTP. They highlight that the L1-I stalls are the dominant factor in the overall stall time followed by the long-latency data misses. Sirin et al. [46] have analyzed Online Analytics Processing (OLAP) workloads. The study has shown that OLAP workloads spend most of the CPU cycles to data cache miss stalls, either due to saturation of the memory bandwidth or long-latency data cache misses. Our experimental methodology while measuring various hardware events using counters on real hardware is very similar to the methodologies of these studies.

Yasin et al. [59] examine Naive-Bayes algorithm and its hardware behavior in an Hadoop execution environment. The study shows that software stack, such as the used JVM, and application code efficiency has a significant impact on the overall performance. Kanev et al. [20] examine collective of machines in a Google datacenter running collective of Google datacenter applications. The study shows that the datacenter workload collection spends most of the time on waiting for dependent data cache accesses due to the data-intensive nature of the datacenter workloads. Beamer et al. [6] presents a graph workload analysis and highlights that graph workloads severely under-utilize the memory bandwidth. These studies are complementary to our work as they focus on workloads with different data and instruction access patterns.

Harizopoulos et al. [16] demonstrate that traditional OLTP systems spend more than half of their execution time within the buffer pool, latching, locking, and logging components. On the other hand, Wenisch et al. [57] and Tozun et. al [53] tie the micro-architectural behavior of the disk-based OLTP into specific code modules by presenting the breakdown of the cache misses into specific code parts of the traditional OLTP software stack at different code granularities.

As Sect. 2.1 explains, the in-memory OLTP systems either remove or simplify most of the traditional disk-based OLTP components. Therefore, the micro-architectural behavior of OLTP workloads when run on disk-based systems cannot be representative for the in-memory systems. Even worse, the previous findings might mislead researchers and developers that aim to improve utilization at the micro-architectural level when running OLTP workloads using in-memory OLTP systems. Therefore, the focus of this paper is to perform a workload characterization study for OLTP benchmarks running on in-memory OLTP systems to understand the low-level differences between in-memory and disk-based OLTP, and based on these findings, provide valuable insights for OLTP systems' design.

3 Setup and methodology

The experiments presented in this paper are executed on real hardware. The rest of this section details the setup and methodology for our study.

Hardware We run experiments on a modern commodity server with Intel's Broadwell processors. Table 1 shows the architectural details of this server. To collect numbers about various hardware events and break down the time spent in specific code modules, we use Intel VTune Amplifier XE 2018 [17], which provides an API for lightweight hardware counter sampling. We disable hyper-threading and turbo-boost to obtain more precise hardware sampling values and increase predictability in measurements.

OS & Compiler We use Ubuntu 16.04.6 LTS and gcc 5.4.0 on the Broadwell server.

Benchmarks We run two types of benchmarks: micro-benchmarks and TPC benchmarks [1]. Our goal is to perform sensitivity analysis and have a more detailed understanding of the systems using the micro-benchmark, while the experiments using the TPC benchmarks serve to give an idea about the behavior of the systems when running well-known real-world applications.

The micro-benchmark uses a randomly generated table with two columns (`key` and `value`) of the type Long. It has two versions: read-only and read-write. The read-only version reads N random rows from the table, whereas the read-write version updates N random rows. Both versions use an index lookup operation on the randomly picked `key` value to reach the row to be read or updated. We also use a modified version of the micro-benchmark where we use strings of 50 bytes for both columns to quantify the impact of data type on micro-architectural utilization in Sect. 6.2.

As for the TPC benchmarks, we use TPC-B and TPC-C. We omit the TPC-E benchmark since recent workload characterization studies demonstrate that TPC-E exhibits similar micro-architectural behavior to the TPC-B and TPC-C benchmarks [12,54].

Analyzed systems We analyze three in-memory OLTP systems: the in-memory OLTP engine of a closed-source commercial vendor (*DBMS M*), an open-source commercial OLTP system (*DBMS N*), and an open-source OLTP engine (*Silo* [55]).

We pick these three systems as they are well known in the community and their design characteristics represent a good variety among today's in-memory OLTP systems. While *DBMS M* adopts multi-versioned concurrency control, *DBMS N* uses physical data partitioning, and *Silo* uses optimistic concurrency control. *DBMS M* implements both hash index and a variant of cache-conscious B-tree index similar to [28,29]. *DBMS N* uses a variant of a self-balancing binary search tree, red-black tree. *Silo* implements Masstree, a highly parallel in-memory index structure [32]. For *DBMS*

Table 1 Server Parameters

Processor	Intel(R) Xeon(R) CPU E5-2680 v4 (Broadwell)
#Sockets	2
#Cores per socket	14
Hyper-threading	Off
Turbo-boost	Off
Clock Speed	2.40 GHz
Bandwidth (per socket)	66 GB/s
L1I / L1D (per core)	32 KB / 32 KB 16-cycle miss latency
L2 (per core)	256 KB 26-cycle miss latency
LLC (shared)	35 MB 160-cycle miss latency
Memory	256 GB

M, we use the B-tree index. Moreover, *DBMS M* use transaction compilation techniques for the stored procedures, whereas *DBMS N* and *Silo* do not. We evaluate *DBMS M* always having its transaction compilation feature turned on in our analyses, except in Sect. 6.1, where we evaluate the transaction compilation feature.

In order to gain better insights about the differences between the in-memory and disk-based OLTP systems, we also include two disk-based systems: a popular, commercial system (*DBMS D*) and the open-source *Shore-MT* [43] storage manager.

To implement benchmarks, we use the SQL frontend of the commercial systems, *DBMS D*, *DBMS M*, and *DBMS N*, and *Silo*'s benchmarks in C++ and *Shore-MT*'s Shore-Kits suite that provides an environment to implement benchmarks for *Shore-MT* in C++.

For all the systems, we use asynchronous logging. Therefore, there is no delay due to I/O in the critical path of the transaction execution.

Measurements We populate the databases from scratch before each experiment and the data remains memory-resident throughout the experiment. In the following sections, we indicate the database sizes used in each experiment before discussing the results. In our experiments, both the database server process executing the transactions and the client processes generating the transactions run on the same machine. We first start the server process, populate the database, and then start the experiment by simultaneously launching all clients that generate and submit transactional requests to the database server.

We profile the database server process by attaching VTune to it during a 120-second benchmark run following a 60-second warm-up period. We repeat every experiment three times and report the average result.

In terms of micro-architectural efficiency, our goal is to observe how well each system exploits the resources of a single core regardless of the parallelism in the system. Therefore, all the experiments except for the ones in Sects. 7 and 8 use a single worker thread executing the transactions.

The choice of a single worker thread also eliminates contention due to several threads trying to access the shared data in the case of non-partitioned systems and distributed transactions in the case of partitioning-based systems. This way we avoid possible misleading micro-architectural conclusions. For example, high contention for a shared data page could lead to multiple threads spinning on a latch for that data page, thus artificially increasing the cache hit ratio.

We use one client to generate request in the single-threaded experiments. *Shore-MT*, *DBMS D*, *Silo*, and *DBMS M* assign one worker thread per client. *DBMS N*, on the other hand, generates one worker thread per data partition, so we configure it to have only one partition. From VTune, we filter the hardware counter results particularly for the identified worker thread excluding the other threads that are responsible for background tasks, e.g., communication between the server and client, parsing transactions, etc.

In multi-threaded experiments (Sects. 7 and 8), we use multiple clients to generate requests for all systems. For *DBMS N*, we also use multiple data partitions and ensure that all transactions access only a single partition. For each system, we gradually increase the number of clients, and profile the execution with the number of clients that give the highest aggregate throughput. From VTune, we filter hardware counter results for each worker thread separately and report their average.

VTune We use Intel VTune 2018. We use VTune's built-in general-exploration analysis for the breakdown of the CPU cycles per the Top-down Micro-architecture Analysis Method. We use VTune's built-in memory-access analysis to measure the consumed memory bandwidth. As we numa-localize our experiments on a single socket, we report average bandwidth per-socket values. We use VTune's built-in advanced-hotspots analysis to perform function call trace breakdown.

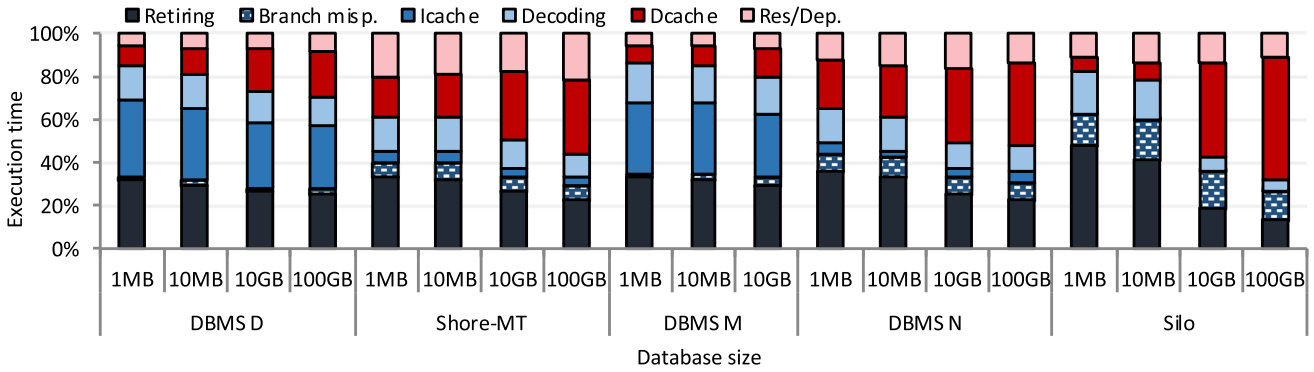


Fig. 1 Breakdowns of the CPU cycles as we increase the database size when running the read-only micro-benchmark

VTune's general-exploration provides the breakdown of the CPU cycles [49,58]. We use a simplified version of VTune's original CPU cycles categorization to be able to interpret the results more easily. We follow the same simplified categorization used in our previous work [46]. In Table 18, Section C of the "Appendix", we provide how each individual CPU cycles component that VTune reports maps to the CPU cycles category that we use.

Each CPU pipeline slot is categorized into one of two components: retiring and stalling. A retiring cycle is a cycle where the processor finishes the execution of an instruction, i.e., retires an instruction. A stalling cycle is a cycle where the processor has to wait, i.e., stall, (e.g., to perform a read from the caches). In an ideal scenario, all CPU cycles would be retiring. Stalling cycles can be further decomposed into five components: (i) branch misprediction, (ii) Icache, (iii) decoding, (iv) Dcache, and (v) resource/dependency. Today's processors use a hardware unit called branch predictor; it predicts the outcome of a branch instruction (i.e., an if() statement) and speculatively executes instructions per the predicted branch direction and/or target. If the processor then realizes the prediction is not correct, it undoes whatever it has been doing and starts executing the correct set of instructions. This cost is defined as the branch misprediction and can be very costly, as it requires canceling a large amount of work. Icache defines the cost of instruction cache and instruction translation lookaside buffer misses. Decoding defines the cost of sub-optimal micro-architectural implementation of the instruction decoding unit. Dcache defines the cost of data-cache misses. Resource/dependency defines the cost of executing instruction that has resource and/or data dependencies. For example, if two instructions require using the same arithmetic-logic unit, one has to wait for the other. This time is identified as the resource dependency time. Or, if an instruction's operand depends on the result of another instruction, the instruction with the dependent operand has to wait for the other instruction to finish. This time is identified as the data dependency time.

4 Micro-benchmark

Before performing an analysis using the community standard TPC benchmarks, we devise a sensitivity study using the micro-benchmark. The goal of this study is to answer the following questions:

- Where do CPU cycles go when running in-memory OLTP? Are they wasted on memory stalls or used to retire instructions?
- Where do memory stalls come from? Are they mainly due to instructions or data for in-memory OLTP?
- What is the impact of the database size on the above metrics?
- Does the amount of work done per transaction affect the results and, if yes, how?

To answer these questions, we break the analysis into two parts. The first part (Sect. 4.1) varies the database size by varying the number of rows in the table while keeping the amount of work done per transaction constant. On the other hand, the second part (Sect. 4.2) varies the amount of work done per transaction by increasing the number of rows read in a transaction while keeping the database size constant.

4.1 Sensitivity to data size

To investigate the impact of database size on the micro-architectural behavior, we populate databases of size 1MB, 10MB, 10GB, and 100 GB. *DBMS M* has a 32 GB of data size limitation. Hence, we use maximum of 10 GB of database for *DBMS M*. The micro-architectural behavior of *DBMS M* does not significantly change as the data size varies (see Fig. 1 and Table 2). Hence, we rely on 10 GB of data to interpret *DBMS M*'s micro-architectural behavior for a large data size. Then, we collect hardware events as the systems run the micro-benchmark with a single transaction type that just reads/updates one random row after an index probe operation. While the results for the read-only version of the micro-

Table 2 Normalized throughput as we increase the database size

	1 MB	10 MB	10 GB	100 GB
DBMS D	1	1	1	1
Shore-MT	3.0	2.9	2.1	1.6
DBMS M	2.8	3.0	3.0	–
DBMS N	7.8	7.8	6.4	3.9
Silo	75.4	62.9	27.5	19.3

benchmark are in the following subsections, the results for the read-write version of the micro-benchmark are in Section A.1 of the "Appendix".

4.1.1 CPU cycles breakdown

Figure 1 shows the CPU cycles breakdowns as the database size is increased. The retiring cycles ratios are similar for databases of sizes 1 MB and 10 MB since the data working set mainly fits in the last-level cache (LLC). As we increase the database size to 10 GB and 100 GB, the retiring cycles ratios are decreased since the data working set no longer fits in caches and the long-latency data misses become more significant. All the retiring cycles ratios are less than 30% for a database of size 10 GB and 100 GB for all the systems. Hence, in-memory OLTP systems spend most of their CPU cycles to stalls similar to disk-based systems.

Shore-MT is bound by Dcache and resource/dependency stalls. *Shore-MT* is a disk-based system known to have a large and complex instruction footprint. Existing work on *Shore-MT* has long shown that *Shore-MT* is Icache-stalls-bound [48,49,53,54]. All machines used in the existing work is Intel's, version 2, Ivy Bridge micro-architecture. We, on the other hand, use a later-generation version 4 micro-architecture, Broadwell, which is the slightly improved version of the version 3 Haswell micro-architecture. Intel has announced an important micro-architectural improvement on the instruction fetch unit of the Haswell micro-architecture [14]. As Hammarlund et al. [14] specifies, "State-of-the-art advances in branch prediction algorithms enable accurate fetch requests to run ahead of micro-operation supply to hide instruction TLB and cache misses.". Hence, instruction fetch unit keeps supplying instructions even though there is an instruction cache miss, which allows overlapping the instruction cache miss latency with useful work. As a result, *Shore-MT*, being a long-standing-Icache-stalls-bound system, has become Dcache- and resource/dependency-stalls-bound. We compare Ivy Bridge and Broadwell in Sect. 10.

The instruction footprint of disk-based systems is large and complex due to the complex relationships among various individual components such as buffer manager and lock manager. *Shore-MT* not suffering from Icache stalls shows that today's processors are powerful enough to mitigate the Icache stalls caused by the large and complex the instruction footprint of disk-based systems. We summarize our findings on Icache stalls in terms of database software and hardware development in Sect. 11.

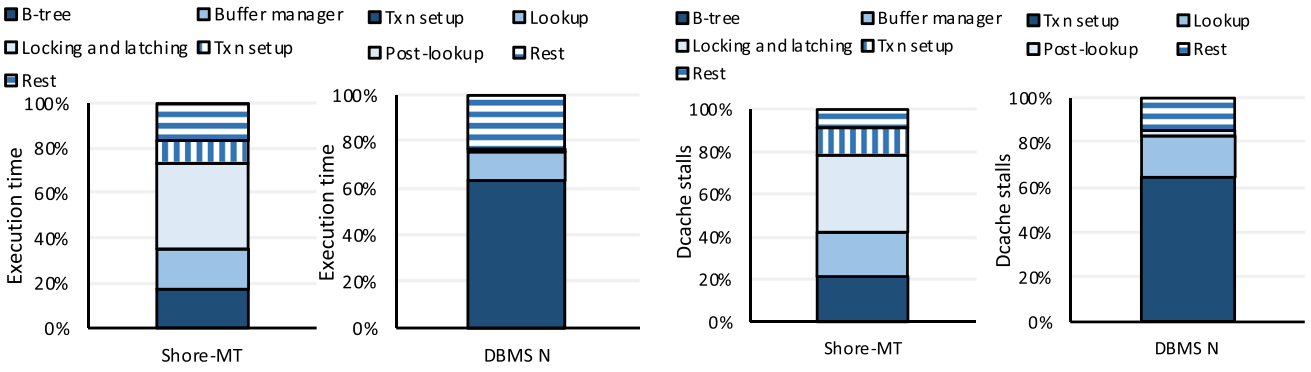
Shore-MT spends 40% of its time on Dcache and resource/dependency stalls even when the data size is small. We examine *Shore-MT*'s call stack and observe that most of the Dcache and resource/dependency stalls are due to buffer manager and centralized locking & latching operations such as looking into the hash table that keeps track of the buffer pool pages and acquiring a lock. As the data size exceeds the LLC size, *Shore-MT* becomes more and more Dcache-bound due to its working set not fitting into the LLC and suffering from expensive LLC misses.

DBMS D and *DBMS M* suffer from high Icache stalls. *DBMS D*, despite being a popular commercial, disk-based OLTP system, relies on a legacy codebase whose instruction footprint is large and complex. *DBMS M*, being the in-memory OLTP engine of *DBMS D*, borrows legacy code modules from *DBMS D*. As a result, its instruction footprint is also large and complex, and it mainly suffers from the Icache stalls. Nevertheless, *DBMS M*'s throughput is three times of *DBMS D* (see Table 2). Hence, the optimizations *DBMS M* adopts help in reducing the instruction and data footprints of *DBMS D*. Nevertheless, both *DBMS D* and *M* mainly suffer from Icache stalls showing the severe effect of using legacy code modules both for disk-based and in-memory OLTP systems.

Unlike *DBMS M*, *DBMS N* does not suffer from Icache stalls. This is because *DBMS N* is a ground-up designed in-memory system. It does not rely on legacy code modules as *DBMS M* does. As a result, its instruction footprint is smaller and less complex than *DBMS M*, and hence, it does not suffer from Icache stalls.

Our previous work [48,49] shows that VoltDB, a similar ground-up designed system to *DBMS N*, is instruction-bound rather than data-bound as we present here. The reason is, once again, the improved micro-architecture of the machine (Broadwell) we use compared to the machine used by [48,49] (Ivy Bridge). The improved instruction fetch unit of Broadwell eliminates the Icache stalls and makes *DBMS N* data-bound. Section 10 discusses this issue in more detail. We summarize our findings on Icache stalls in terms of database software and hardware development in Sect. 11.

DBMS N is an in-memory system, which eliminates the heavy disk-based system components such as buffer pool and lock manager. Nevertheless, it suffers significant amount of Dcache and resource/dependency stalls for small data sizes, similar to *Shore-MT*. We examine *DBMS N*'s call stack and observe that the Dcache and resource/dependency stalls are largely due to the cost of setting up and instantiating the transactions. As the data size is increased, *DBMS N* becomes more



(a) Execution time breakdown. **(b)** Dcache stalls breakdown.

Fig. 2 Execution time and Dcache stalls breakdown for *Shore-MT* and *DBMS N* for 1 MB of database size

and more Dcache-bound due to its working set not fitting into the LLC and suffering from expensive LLC misses.

Silo has high retiring cycles for small data sizes. *Silo* is an in-memory system eliminating the costly buffer manager and centralized locking & latching components that disk-based systems use. Moreover, *Silo* is a kernel OLTP engine that has transactions hard-coded in C++. Hence, it does not suffer from the cost of instantiating and scheduling user request as *DBMS N* does. As a result, it does not suffer from Dcache stalls when the data size is small, and it has high retiring cycles ratio for small data sizes. As the data size exceeds the LLC size, *Silo* becomes Dcache-bound such that *Silo*'s retiring cycles are the lowest among all the systems.

In addition to the Dcache stalls, *Silo* also suffers from significant amount of branch misprediction. This is due to the in-node searches that *Silo* performs during the index traversal [1]. While all the systems perform an index traversal during their transaction processing, it only surfaces up on *Silo* thanks to its lean codebase and efficient data structure it uses. Other systems suffer from other overhead such as complex instruction footprint as in *DBMS D* and *M*, large data footprint as in *Shore-MT*, or inefficient index structure as in *DBMS N*.

We observe that all the systems suffer 10–15% decoding stalls. Decoding stalls are the stalls due to the inefficiencies in the micro-architectural implementation of the instruction decoding unit of the processor. As Intel relies on a Complex Instruction Set Computer (CISC) type of microprocessor design, it requires decoding instructions into micro-operations. It is known that Intel's instruction decoding unit is a legacy micro-architecture and has several penalties [18]. To avoid these penalties, Intel has introduced a Decoded Stream Buffer (DSB), which is a micro-operation cache inside the pipeline that allows side-stepping the decoding unit and providing already decoded instructions to the processor. However, DSB is small (1.5 KB). If the workload's instruction working set does not fit into the DSB, it has to pass through the legacy decoding unit and suffer from the penalties of the legacy decoding unit. As OLTP workloads' instruction footprint is large and complex, they pass through the legacy decoding unit and suffer the decoding stalls. Nevertheless, the caused penalties are relatively small, 10-15%, and hence do not constitute a significant problem.

4.1.2 Throughput

Table 2 shows normalized throughputs for each system as the database size is increased. The throughput values are normalized to *DBMS D* for each database size individually. The relative performance between *DBMS D* and *M* remains stable. This is because both *DBMS D* and *M* are instruction-bound. Hence, the increased data size does not significantly affect their performance. *Shore-MT*, *DBMS N*, and *Silo*'s relative performance, on the other hand, is decreased as the data size is increased. This is because *Shore-MT*, *DBMS N* and *Silo* are data-bound, i.e., Dcache and resource/dependency-stalls-bound. The increased data size results in a more substantial drop in their throughput than *DBMS D* and *M*. As a result, their throughputs get close to *DBMS D* and *M* as the data size increases.

All the in-memory systems are faster than the disk-based systems for all the data sizes. This shows that the optimizations that in-memory systems implement significantly help improving the throughput. *DBMS M*, despite its large and complex instruction footprint, is faster than *Shore-MT* for 10 GB of database. As *Shore-MT* mainly suffers from Dcache stalls, this shows the severe negative effects of disk-based systems' data overhead, such as large index and buffer pool pages.

DBMS N is faster than *DBMS M* thanks to its smaller and simpler codebase. *DBMS N* is a ground-up designed system. Hence, it does not borrow any legacy codebase as *DBMS M* does. As a result, it does not suffer from Icache stalls, and it

[1] Silo uses linear search as its in-node search algorithm.

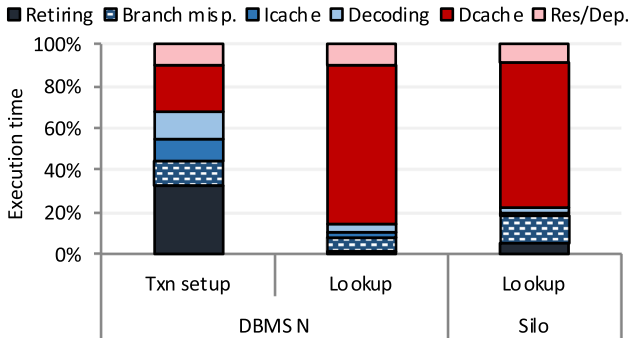


Fig. 3 Breakdowns of the CPU cycles for Transaction setup and Index lookup for *DBMS N* and *Silo*

is significantly faster than *DBMS M*. *Silo* is the fastest system we have profiled. One reason for that is *Silo* is a kernel OLTP engine, which does not suffer from the cost of setting up and instantiating the transactions. *DBMS N*, on the other hand, is an end-to-end SQL-based OLTP system. Another reason is that *Silo* uses an efficiently implemented index structure, Masstree [32]. Hence, its data footprint is also succinct. As a result, it is 4.9x faster than *DBMS N* for 100 GB of database. We examine the inefficient index structure issue of *DBMS N* in Sects. 4.1.3 and 4.2.1 in more detail.

4.1.3 DBMS N versus Silo

Silo is $\sim 5\times$ faster than *DBMS N* as shown by Table 2 for 100 GB of database. We examine the call stack of *DBMS N* and *Silo*. *DBMS N* spends $\sim$ 33% of its time while setting up and instantiating the transactions, from which *Silo* minimally suffers due to being a kernel OLTP engine. Hence, 33% of the $5\times$ difference is due to the transaction setup and instantiation work that *DBMS N* performs. These are functions like `processInitiateTask()`, `xfer()` (dequeues user requests) and `coreExecutePlan Fragments()`.

DBMS N spends most of the remaining 67% of its time in index lookup. In particular, *DBMS N* spends $\sim$44% of its time in index lookup and $\sim$19% of its time in various small-sized functions each taking less than 1% of the execution time. *Silo* spends $\sim$75% of its time in index lookup. Therefore, most of the remaining 67% of the $5\times$ throughput difference between *DBMS N* and *Silo* is due to *Silo* using a more efficient index structure than *DBMS N*.

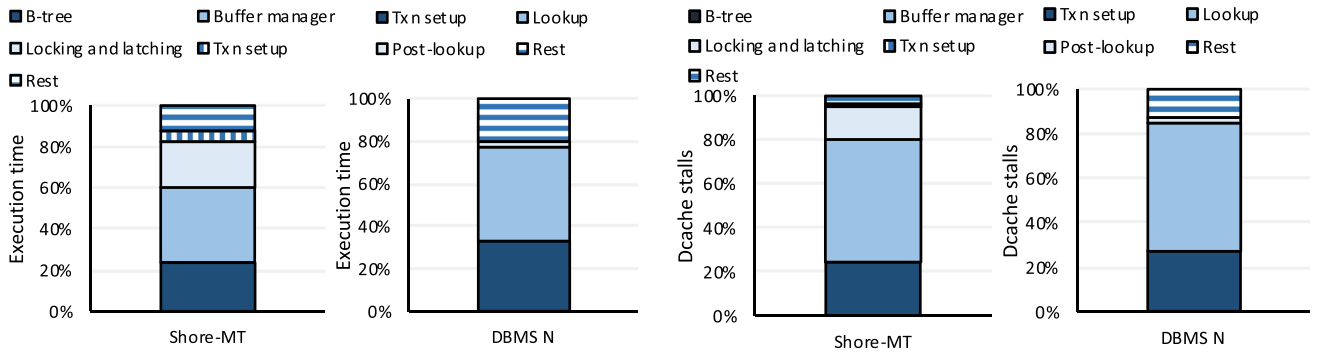
We examine the micro-architectural behavior of the transaction setup and index lookup components of *DBMS N* separately in Fig. 3 for 100 GB of database. The figure shows that transaction setup component only modestly suffers from Dcache stalls, whereas the lookup component is exclusively Dcache-stalls-dominated. The overall micro-architectural behavior of *DBMS N* is the composition of these two micro-architectural behavior. We also examine the micro-architectural behavior of the index lookup component of *Silo*. Similar to *DBMS N*, the micro-architectural behavior of the index lookup operation of *Silo* is exclusively dominated by Dcache stalls. As *Silo*'s execution time dominated by the index lookup operation, 67% of the $5\times$ throughput difference between *DBMS N* and *Silo* is due to the used index structure. Therefore, *DBMS N* can significantly improve its performance by using a more efficient index structure. We describe *DBMS N* and *Silo*'s used index structures in Sect. 4.2.1 in more detail.

4.1.4 *Shore-MT* versus *DBMS N*

Shore-MT and *DBMS N* spend a significant portion of their time in Dcache stalls for small data sizes (as shown by Fig. 1). We examine the execution time and Dcache stalls breakdown for *Shore-MT* and *DBMS N* for small and large data sizes. We identify five components for *Shore-MT* at the software level: (i) B-tree, (ii) buffer manager, (iii) locking and latching, (iv) transaction setup, and (v) rest. We identify four components for *DBMS N* at the software level: (i) Transaction setup, (ii) Index lookup, (iii) Post-lookup, and (iv) Rest. Figure 2 shows the results for 1 MB of data.

Shore-MT spends most of the execution time processing disk-based system components such as buffer manager and centralized locking & latching components. Similarly, most of *Shore-MT*'s Dcache stalls are coming from the locking & latching and buffer manager components. *DBMS N* is an in-memory system. It does not use a buffer manager component, and it uses a partitioning-based, lightweight concurrency control mechanism. However, as it is a real-life OLTP system, it performs the necessary work to setup and instantiate transactions using functions like `processInitiateTask()`, `xfer()` (dequeues user requests) and `coreExecutePlanFragments()`. *DBMS N* spends most of the execution in transaction setup, and most of *DBMS N*'s Dcache stalls are coming from the transaction setup component.

We also examine *Shore-MT* and *DBMS N*'s execution time and Dcache stalls breakdowns for 100 GB of database. Figure 4 shows the results. *Shore-MT*, being a disk-based system, spends most of the execution time in buffer manager and locking & latching components. However, unlike it is for 1 MB of database, buffer manager component consumes a significantly larger portion of the execution time than the locking & latching component. As the data size is increased, there are larger and larger number of buffer pool pages. Hence, the data footprint is increased, and the amount of time spent inside buffer manager is increased. Similarly, *Shore-MT*'s Dcache stalls are mostly due to the buffer manager components for 100 GB of database as shown by Fig. 4b. Our findings for *Shore-MT* corroborates with the findings of Harizopoulos et al. [16].



(a) Execution time breakdown. **(b)** Dcache stalls breakdown.

Fig. 4 Execution time and Dcache stalls breakdown for *Shore-MT* and *DBMS N* for 100 GB of database size

DBMS N spends significantly less time on setting up the transactions for 100 GB of database compared to 1 MB of database. As the data size is increased, the amount of work that the transactions perform is increased. As a result, transaction setup time is reduced from ∼60% to ∼33%, and the index lookup time is increased from ∼10% to ∼44%.

Shore-MT's disk-based system overhead is persistent across different data sizes due to the fundamental architectural reasons. Most of the execution time is spent inside the disk-based system components such as locking & latching and buffer manager, although the system component that consumes the largest portion of the execution time shifts from the locking & latching to the buffer manager component as the data size is increased from 1 MB to 100 GB.

DBMS N's transaction setup overhead depends on the data size. As the data size is increased, the amount of work that transactions perform is increased. As a result, the transaction setup consumes smaller and smaller portion of the execution time as the amount of work per transaction is increased.

Therefore, *Shore-MT* requires a major architectural redesign to reduce the data footprint and deliver as high throughput as in-memory OLTP systems deliver. *DBMS N* spends the large portion of its execution time inside the OLTP engine for large data sizes. Hence, it can highly benefit from optimizing internals of the OLTP engine such as using a more efficient index structure as shown in Sect. 4.1.3.

4.1.5 Summary

Relative throughput of OLTP systems widely vary among different categories of OLTP systems. However, CPU cycles utilization of all the OLTP systems are low. *DBMS D* and *M*, relying on legacy codebases, suffers from Icache stalls. *Shore-MT*, *DBMS N* and *Silo*, being either OLTP kernels having transactions hard-coded in C++, and/or a ground-up designed in-memory system, eliminate the Icache stalls. The reduced Icache stalls cause Dcache stalls to surface, which *Shore-MT*, *DBMS N* and *Silo* suffer from. The Dcache stalls are largely due to the random-data-access nature of the workload, in addition to the cost of buffer manager and locking & latching overhead for *Shore-MT* and the small cost of transaction setup for *DBMS N*. As a result, *Shore-MT*, *DBMS N* and *Silo* spend only 30% of the CPU cycles for retiring instructions similar to *DBMS D* and *M*.

4.2 Sensitivity to work per transaction

To investigate the impact of the amount of work per transaction on the micro-architectural behavior, we increase the number of rows that a transaction accesses from 1 to 10 and then to 100. We perform these experiments with 100 GB dataset for all the systems except *DBMS M*. We use 10 GB of database for *DBMS M* due to its 32 GB of maximum database size limitation. In the following sub-sections, we present the results for the read-only version of the micro-benchmark. The results for the read-write version of the micro-benchmark can be found in Section A.2 of the "Appendix".

Figure 5 shows the CPU cycles breakdowns as the amount of work per transaction is increased. *DBMS D* and *M* suffer less and less from Icache stalls as we increase the amount of work per transaction. The repetitive behavior within a transaction leads to a better instruction cache locality. As a result, the code for the other layers of the system that surround a transaction's execution (e.g., the code outside the storage manager) is executed less frequently since the transactions get longer as we increase the amount of work done per transaction. For example, where probing 100 rows per transaction stresses purely the storage manager component, probing 1 row also stresses the other layers such as query parsing, work done while starting/ending a transaction, etc. As a result, Icache stalls are decreased as the amount of work per transaction is increased.

DBMS D and *M*'s Dcache stalls are increased as we increase the work done per transaction. As we read more

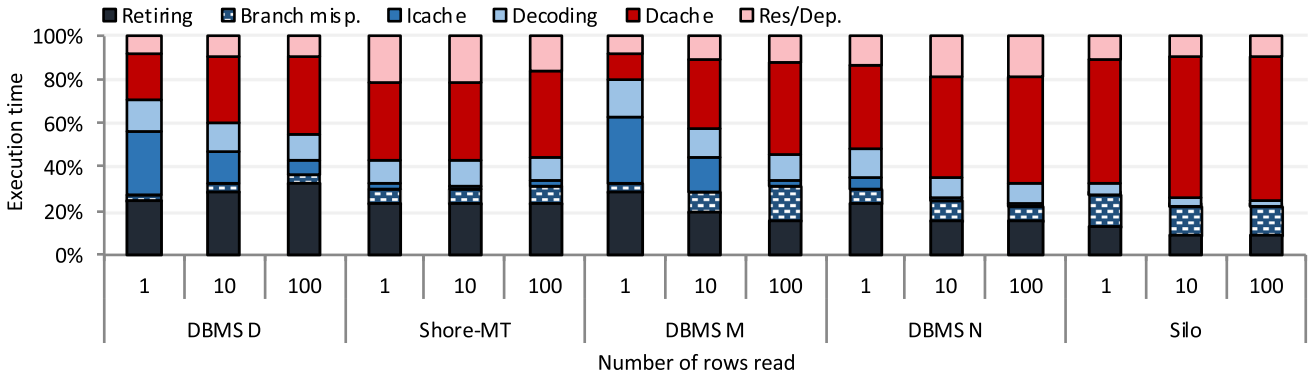


Fig. 5 Breakdowns of the CPU cycles as we increase the amount of work done per transaction with a database of size 100 GB (10 GB for DBMS M)

random rows per transaction, we make more frequent random data accesses, which leads to a higher data miss rate and hence higher Dcache stalls.

Shore-MT, *DBMS N* and *Silo* have slightly increased Dcache stalls as the amount of work per transaction is increased. *Shore-MT*, *DBMS N* and *Silo*'s instruction footprint is small and simple enough when the number of rows read per transaction is 1. Hence, the increased instruction locality does not make a significant difference in terms of their micro-architectural behavior.

Table 3 shows the normalized throughputs. *DBMS D* and *DBMS M* are instruction-stalls-bound as shown in the previous section. Therefore, their delivered throughput does not drop significantly as the data size is increased. *DBMS D*'s normalized throughput for 1MB, 10MB, 10GB, and 100 GB is: 1, 0.93, 0.84, and 0.81. The throughput of *DBMS D* drops only by 3% as the data size is increased from 1 MB to 100 GB. Similarly, *DBMS M*'s normalized throughput for 1MB, 10MB, and 10 GB is: 1, 1.01, and 0.89. *DBMS M*'s throughput drops by only 11% as the data size is increased from 1 MB to 10 GB. Therefore, we assume that *DBMS M*'s throughput for 100 GB is similar to its throughput for 10GB, and we use *DBMS M*'s throughput for 10 GB in Table 3.

DBMS M's relative throughput is increased as the number of rows per transaction is increased. *DBMS M* becomes even faster than *DBMS N* as the number of rows per transaction is increased to 10 and 100. This because of the increased instruction locality that *DBMS M* benefits from. As the number of rows read per transaction is increased, *DBMS M* executes the in-memory OLTP engine more and more. As a result, it suffers less and less from the legacy codebase that it borrows from *DBMS D*. This shows that the core in-memory OLTP engine of *DBMS M* is efficiently implemented and benefits from the in-memory systems optimizations.

Shore-MT, *DBMS N*, and *Silo*'s relative throughput to *DBMS D* is decreased as the number of rows per transaction is increased. This is because *DBMS D* benefits from the increased instruction locality as the number of rows is increased.

Table 3 Normalized throughput as we increase the amount of work done per transaction with a database of size 100 GB (10 GB for *DBMS M*)

	1 row	10 rows	100 rows
DBMS D	1	1	1
Shore-MT	1.6	0.7	0.7
DBMS M	3.0	3.7	3.9
DBMS N	3.9	2.1	1.7
Silo	19.3	8.0	6.0

DBMS N's relative throughput with respect to *Silo* is decreased as the number of rows is increased. *DBMS N*'s relative throughput with respect to *Silo* is: 4.9, 3.8, and 3.5 for 1, 10, and 100 rows, respectively. This is because *DBMS N* executes less and less the code to setup and instantiate the transactions as the amount of work per transaction is increased. As a result, its throughput gets closer and closer to *Silo*, which minimally suffers from the work required to setup and instantiate the transactions. Nevertheless, *Silo* is 3.5× faster than *DBMS N* even when *DBMS N* largely eliminates the transaction setup overhead. We examine this throughput difference in more detail in the following section.

4.2.1 Code modules breakdown

To better understand the impact of legacy code, as well as components outside the storage manager, we quantify the percentage of the execution time spent in the OLTP engine as the amount of work per transaction is increased for the disk-based system *DBMS D*, and in-memory systems *DBMS M* and *DBMS N*. While performing this breakdown, we have done a best-effort categorization based on the code modules reported by VTune as part of the worker thread execution of each system. Figure 6 shows the results.

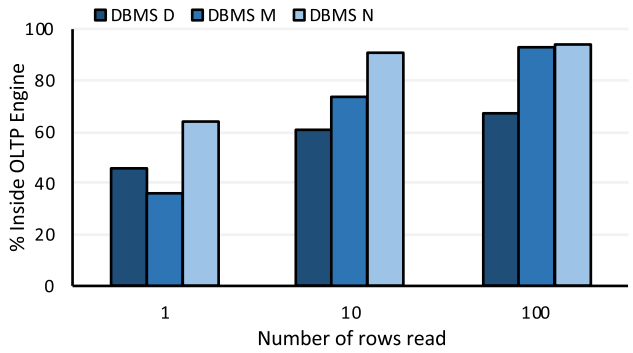


Fig. 6 The percentage of the time spent inside the OLTP engine as we increase the amount of work done per transaction with a database of size 100 GB for *DBMS D* and *DBMS N*, and 10 GB for *DBMS M*

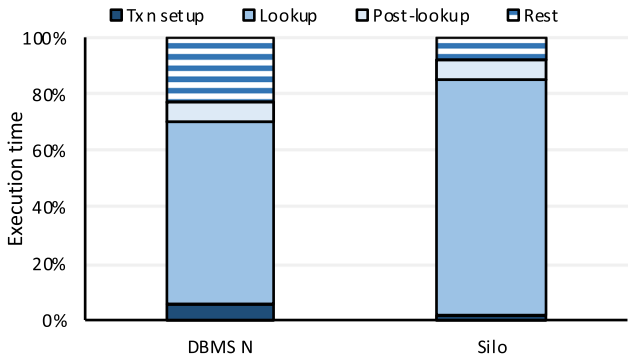


Fig. 7 Function call trace breakdown for *DBMS N* and *Silo* when running the micro-benchmark that reads 100 rows per transaction

We observe that *DBMS D* and *M* systems spend only 35 to 45% of their time inside OLTP engine showing the dominance of the legacy code overhead both systems suffer. The amount of time spent inside the OLTP engine increases as the number of rows read increases. This increase is less pronounced for the disk-based system *DBMS D* showing the higher overhead of the code outside the OLTP engine. For *DBMS M*, when we increase the number of rows from 1 to 10, the percentage inside the OLTP engine is significantly increased showing the reduced effect the legacy code overhead that *DBMS M* borrows from the traditional disk-based OLTP system it belongs to. These results explain better why *DBMS D* and *M*'s relative throughput (shown in Table 3) gets higher and higher as the number of rows read per transaction is increased.

DBMS N is a ground-up designed in-memory system. As a result, it does not suffer from a legacy codebase as *DBMS D* and *M* do. However, as being an end-to-end system, it spends certain amount of time to setup, instantiate and finalize the transactions. In Fig. 6, the amount of time that *DBMS N* does not spend inside the OLTP engine corresponds to this instantiation and finalization. This time is ∼33% for reading 1 row per transaction since the transaction is short. As the number of rows per transaction increases to 10 and 100, the amount of time spent for setting up and instantiating the transactions is reduced to ∼10% and 5%, respectively. This shows that, depending on the amount of work per transaction, the work required for setting up and finalizing the transaction can be significant.

On the other hand, despite the large amount of work per transaction and reduced overhead of transaction setup and finalization, *DBMS N* is still 3.5× slower than *Silo* for 100 rows per transaction (see Table 3). We break the execution time of *DBMS N* and *Silo* down to their function call stack to see this difference better. We use the same execution time breakdown at the software level used for *DBMS N* in Sect. 4.1.3. Figure 7 shows the results for 100 rows.

Both systems spend most of their time performing the index lookup. Therefore, the main reason for the performance difference between *DBMS N* and *Silo* is the inefficient index structure of *DBMS N*. We examined *DBMS N*'s index data structure. We saw that *DBMS N* uses red-black tree as its index structure. Red-black trees are self-balancing binary search trees, where the number of elements per node is 1. Hence, at every level of the tree, red-black tree performs a single comparison. As every node of the tree is in a random memory location, red-black tree is subject to a data cache miss at every level during the index traversal.

Silo, on the other hand, uses Masstree, which is a variant of B-tree. As all the B-trees, Masstree's nodes have a particular node size and fanout. It is 15 for *Silo*. Hence, instead of 1, it keeps 15 elements per node. As the tree depth drops exponentially with the node size, *Silo*'s index is much more shallow than *DBMS N*'s red-black tree. Therefore, Masstree is subjected to a significantly less number of data cache misses. Furthermore, Masstree software prefetches the node's data blocks by injecting a software prefetch instruction during the index traversal. As a result, *Silo* is subject to a single data cache miss for the entire node it accesses at every level of the tree, making *Silo* significantly faster than *DBMS N*. This shows that, despite the overhead of a real-life system, the efficiency of the used index structure is still the most crucial factor in defining the performance characteristics of an in-memory OLTP system.

Finally, *DBMS N*'s rest component is significantly higher than that of *Silo*. This is because *DBMS N*, being a real-life system, executes more functions to provide the end-to-end response.

5 TPC benchmarks

Section 4 performs a sensitivity analysis using a simple micro-benchmark to gain a fine-grained understanding of the the micro-architectural behavior of the OLTP systems. This

section investigates the behavior of the same systems while running the more complex and community standard TPC-B (Sect. 5.1) and TPC-C (Sect. 5.2) benchmarks. All the experiments in this section use a database of size 100 GB, except *DBMS M* for which we use the maximum allowed database size of 32 GB. Similar to Sect. 4, we analyze the CPU cycles breakdowns and normalized throughput values.

DBMS D and *DBMS M* are instruction-stalls-bound. Therefore, their delivered throughput does not drop significantly as the data size is increased as described in Sect. 4.2, and we assume that *DBMS M*'s throughput for 100 GB is similar to its throughput for 32 GB both for TPC-B and TPC-C.

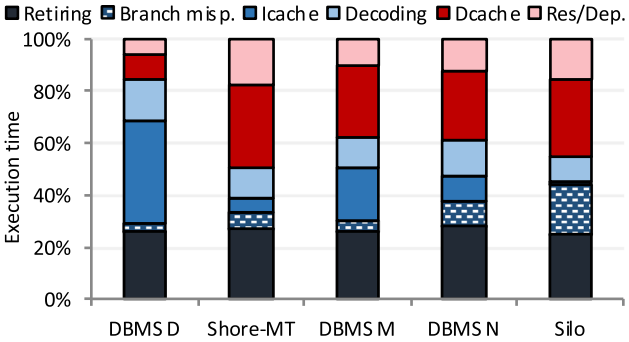


Fig. 8 Breakdowns of the CPU cycles while running TPC-B

5.1 TPC-B

TPC-B is an update-heavy benchmark that simulates a banking system. `AccountUpdate` is its only transaction type, which updates one row each in three tables, `Branch`, `Teller`, and `Account`, and appends a row to the `History` table.

Figure 8 shows the CPU cycles breakdowns and Table 4 shows normalized throughputs for TPC-B. *DBMS D*, *Shore-MT*, *DBMS N*, and *Silo* suffer less from Dcache stalls compared to the micro-benchmark that reads 1 row per transaction (see Fig. 1). This is mainly because TPC-B has a better data locality compared to the micro-benchmark. When running the micro-benchmark, we randomly probe rows from a 100 GB table, which includes more than one billion rows. On the other hand, TPC-B first probes one of the ∼ 20K `Branches` randomly. Then, it probes one of the ∼ 200K `Tellers` and one of the ∼ 2 billion `Accounts`. Finally, it inserts on row into the `History` table. Hence, the probability of re-accessing the same branch or teller as well as the same `History` table page is quite high compared to that of the micro-benchmark's single large table.

DBMS M suffers less from Icache stalls and more from Dcache stalls for TPC-B compared to the micro-benchmark. *DBMS M* relies on a multi-version concurrency control mechanism, where the updates are kept in deltas. Each new version of the data is written to a new memory location called delta, and the pointer to the record is updated to point to the location of the delta. Multiple versions are kept in multiple deltas that are chained one-after-the-other. Occasionally, the deltas are consolidated. As TPC-B is an update-heavy benchmark, *DBMS M* requires creating a new delta for every transactions and hence perform more random data accesses during the traversal and/or consolidation of the delta chain. This results in higher degree of Dcache stalls for TPC-B compared to the read-only micro-benchmark. We confirm this hypothesis by the read-write version of the read-only micro-benchmark discussed in Section A.1 in the "Appendix".

The normalized throughput values follow a similar trend to the micro-benchmark. All the in-memory systems are faster than the disk-based systems. Among the in-memory systems, *DBMS N* is faster than *DBMS M* thanks to not suffering from the legacy code modules, and *Silo* is faster than *DBMS N* thanks to its efficient engine components and not suffering from the cost of setting up and instantiating the transactions.

5.2 TPC-C

After investigating the TPC-B benchmark, this section focuses on the more complex TPC-C benchmark. TPC-C models a wholesale supplier with nine tables and five transaction types (2 of which are read-only and form 8% of the benchmark mix). In terms of the database operations, the TPC-C transactions contain probes, inserts, updates, and joins covering a richer set of operations than TPC-B. Therefore, we expect a different behavior for TPC-C than TPC-B.

Figure 9 shows the CPU cycles breakdowns and Table 4 shows normalized throughputs. The micro-architectural behavior follows a similar trend to the micro-benchmark and TPC-B. While *DBMS D* and *M* are Icache-stalls-bound, *Shore-MT*, *DBMS N*, and *Silo* are Dcache- and resource/dependency-stalls-bound. All the systems suffer less from the Dcache stalls compared to the micro-benchmark thanks to the workload locality that TPC-C has. As a result, they have slightly higher retiring cycles ratio than the micro-benchmark.

The normalized throughput values follow a similar trend to the micro-benchmark. All the in-memory systems are faster than the disk-based systems. Among the in-memory systems, *Silo* is faster than *DBMS M* and *DBMS N*.

DBMS N's relative throughput is less for TPC-C compared to the micro-benchmark and TPC-B. We examined *DBMS N*'s call stack. *DBMS N*'s time spent in serializing/deserializing tuples is significantly increased for TPC-C compared to the micro-benchmark. *DBMS N* keeps every tuple in its own format (byte array) and deserializes/serializes the data during transaction processing. As TPC-C requires in-transaction processing of the tuples, such as incrementing the order ID for the new order transaction, *DBMS N*'s relative

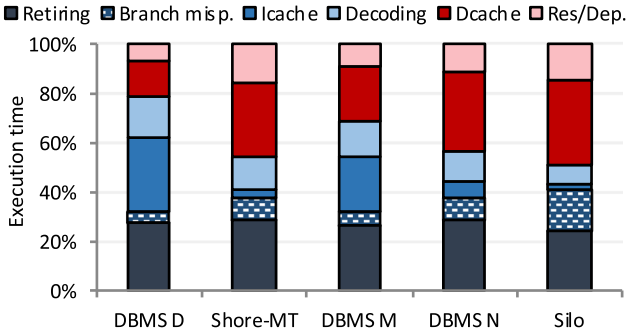


Fig. 9 Breakdowns of the CPU cycles while running TPC-C

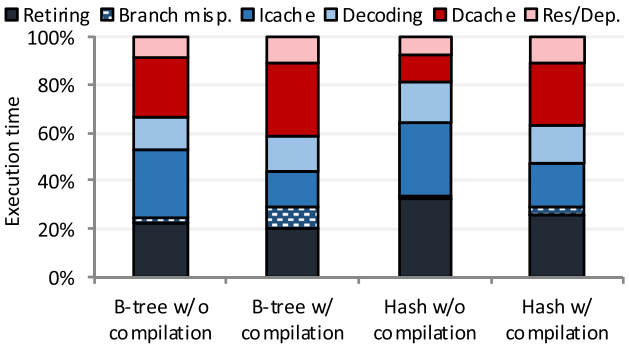


Fig. 10 Breakdowns of the CPU cycles for different index structures with and without compilation optimizations while running the micro-benchmark for *DBMS M*

Table 4 Normalized throughput for TPC-B and TPC-C with a database of size 100 GB (32 GB for *DBMS M*)

	TPC-B	TPC-C
DBMS D	1	1
Shore-MT	1.7	1.2
DBMS M	2.2	3.1
DBMS N	5.4	2.7
Silo	18.5	20.0

Table 5 Normalized throughput for different index structures with and without compilation for *DBMS M*

	Micro-bench.	TPC-C
B-tree w/o comp.	1	1
B-tree w/ comp.	5	5
Hash w/o comp.	1.5	1
Hash w/ comp.	9	5

throughput is decreased when running complex benchmark compared to when running a simple micro-benchmark.

Shore-MT's relative throughput is less for TPC-C compared to the micro-benchmark and TPC-B. We examined *Shore-MT*'s call stack. *Shore-MT*'s time spent on B-tree search and lock manager are significantly increased for TPC-C than for the micro-benchmark. This highlights *Shore-MT*'s index structure and locking & latching processing overhead becoming more prominent for a complex benchmark. This is also visible in Table 3, where *Shore-MT*'s throughput is lower than *DBMS D* for probing 10 and 100 rows per transaction.

6 Index and compilation optimizations, and data types

This section analyzes the impact of index and compilation optimizations the in-memory systems adopt, as well as the impact of the data types, at the micro-architectural level. Among the systems used in this study, *DBMS M* is the only one that allows enabling/disabling the compilation optimizations and using two different index structures: hash index and a variant of cache-conscious B-tree index similar to [28,29]. Therefore, while we use *DBMS M* for analyzing the impact of index and compilation optimizations, we experiment with all the three in-memory systems (*DBMS M*, *DBMS N*, and *Silo*) to quantify the effect of different data types.

6.1 Impact of index type and compilation

To quantify the impact of the type of index and compilation on the micro-architectural utilization, we start with the read-only variant of the micro-benchmark where we access 10 rows per transaction from the 10 GB dataset. Figure 10 presents the CPU cycles breakdowns, and Table 5 presents the normalized throughput values. The results for the read-write version of the micro-benchmark can be found in Section B of the "Appendix". Transaction compilation has a significant effect on the instruction stalls, resulting in $\sim$ 50% reduction in the Icache stalls regardless of the index type. Transaction compilation enables many optimizations in the instruction stream. It can eliminate the virtual function calls. It can inline templated function calls. It can eliminate type checkings and certain branches. Lastly, it allows the compiler to employ its own optimizations more aggressively as it reduces the whole task into a generated code file. As a result, transaction compilation reduces the length and complexity of the instruction footprint.

Table 5 shows the transaction compilation improves the throughput by 5–6×. We observe that B-tree has more Dcache stalls than the Hash index when transactions are not compiled. This is expected as B-tree requires multiple levels of random lookups, whereas Hash index usually requires one or two random lookups. When the transactions are compiled, B-tree and Hash index have similar ratios of Dcache stalls.

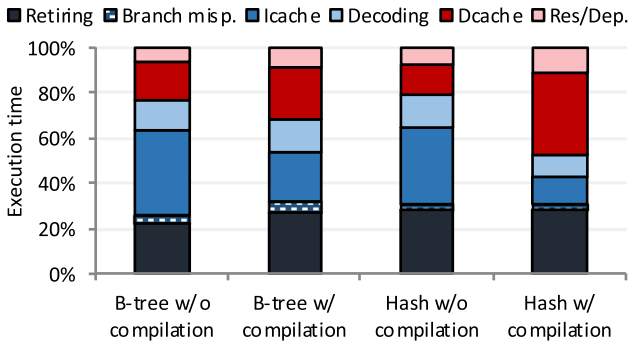


Fig. 11 Breakdowns of the CPU cycles for different index structures with and without compilation while running TPC-C for *DBMS M*

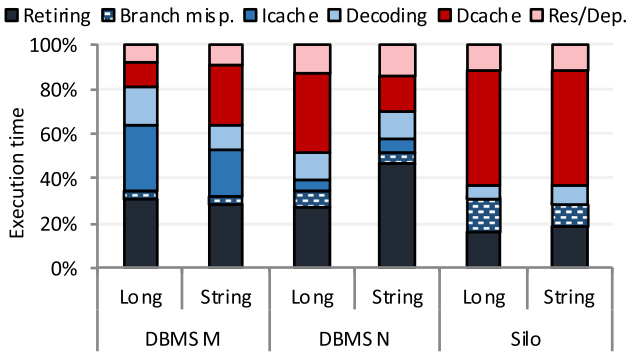


Fig. 12 Breakdowns of the CPU cycles for string and long data types while running the micro-benchmark

However, as Table 5 shows, throughput with Hash index is 80% higher than throughput with B-tree.

We repeat the experiment above using the TPC-C benchmark. Figure 11 shows the CPU cycles breakdown, and Table 5 shows the normalized throughput values. Once again, compilation optimizations reduce instruction stalls significantly for both index types. Moreover, transaction compilation improves *DBMS M*'s throughput by 5× for both B-tree and Hash index types. For Dcache stalls, since the TPC-C benchmark requires fewer random data reads compared to the micro-benchmark, we do not observe a significant difference in Dcache stalls for B-tree and Hash index.

6.2 Impact of data type

To quantify the impact of different data types on micro-architectural utilization, we use the read-only version of the micro-benchmark where we probe 1 row per transaction over a 100 GB database (10 GB for *DBMS M*). The results for the read-write version of the micro-benchmark can be found in Section B of the "Appendix". We modify the micro-benchmark to use two 50 bytes string columns instead of two long columns in the table and compare the two versions.

Figure 12 presents the CPU cycles breakdown. *DBMS M*'s Dcache stalls are higher for string than they are for long. We examined the modules breakdown of *DBMS M* for string and long. We observed that the increased Dcache stalls are due to the legacy string processing code that *DBMS M* borrows from its legacy codebase. As string processing operations usually have high spatial locality, the increased Dcache stalls highlight an inefficient string processing implementation such as carrying high memory overheads for the string objects.

DBMS N suffers less from Dcache stalls for string compared to long. This is expected as string processing operations usually have high spatial locality. We examined *DBMS N*'s function call stack and observed that string comparison code constitutes a larger fraction of the execution time with less Dcache stalls.

Table 6 Normalized throughput for string and long data types while running the micro-benchmark

	Long	String
DBMS M	1	0.7
DBMS N	1	1
Silo	1	0.7

Silo has a similar micro-architectural behavior for long and string data types. This is because *Silo*'s index structure, Masstree, combines B-tree and trie index structures, where every node of the trie structure is a separate B-tree. Masstree slices the keys into pieces of eight bytes and does a separate B-tree search for every eight bytes of the key, while traversing the overall trie structure. As a result, using a long, 50-byte of string key does not make a significant difference in terms of the data access pattern during the key comparisons.

Keeping B-tree within a trie structure allows skipping the upper levels of the trie structure for keys with long common prefixes (such as http URLs). The keys we use do not have such a feature. Hence, Masstree search boils down to multiple levels of B-tree searches.

Table 6 presents the throughput values for the string data type that is normalized to the long data type. The results show that *DBMS M* and *Silo* deliver lower throughput for string than they deliver for long. This is expected as the amount of work required to process the longer-sized string data type is larger than it is for the long data type. *DBMS N*'s throughput remains the same for the long and string data type. This is because, unlike *DBMS M* and *Silo*, *DBMS N* is able to exploit the spatial locality of string processing. As a result, the increased work due to using strings is balanced out with the higher spatial locality of string search.

7 Impact of multi-threading

This section analyzes the effect of running multiple server side threads on the micro-architectural behavior. The single-threaded experiments aim to present an idealized case since it

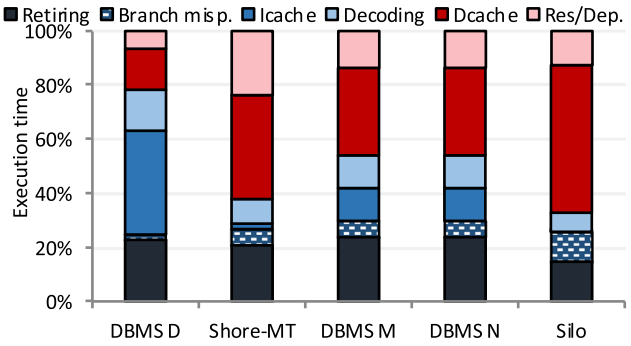


Fig. 13 Breakdowns of the CPU cycles for the multi-threaded experiments while running the micro-benchmark

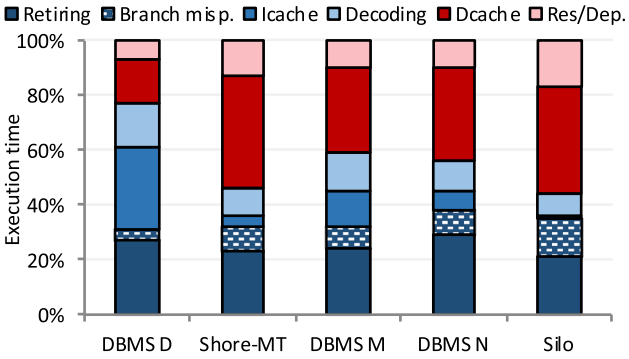


Fig. 14 Breakdowns of the CPU cycles for the multi-threaded experiments while running TPC-C

avoids cache invalidations due to data sharing across different worker threads or misleading artificially high IPC values due to threads spinning under possible contention. On the other hand, multi-threaded experiments aim to investigate a more realistic scenario where systems are loaded with multiple threads executing transactions from multiple clients.

Figures 13 and 14 show the CPU cycles breakdowns while running the read-only version of the micro-benchmark when reading 1 row and TPC-C benchmark, respectively. We use a database of size 100 GB in both of the experiments for all the systems except *DBMS M*. We use 10 GB of database for the micro-benchmark, and 32 GB of database for the TPC-C benchmark for *DBMS M*. We observe that the micro-architectural behavior of the individual systems is similar to the single-threaded executions when running the micro-benchmark. *DBMS D* mainly suffers from the Icache stalls, whereas *Shore-MT*, *DBMS N* and *Silo* majorly suffer from the Dcache stalls. *DBMS M* suffers less from the Icache stalls compared to the single-threaded execution. We examined the code modules breakdown of *DBMS M* and observed that the amount of time it spends inside the OLTP engine remains the same across the single- and multi-threaded executions. However, the amount of time *DBMS M* spends in different modules that are outside the OLTP engine changes significantly across the single- and multi-threaded executions. Hence, the reduced Icache stalls in the multi-threaded execution is likely due to the higher instruction locality of the code modules that are more active during multi-threaded execution.

Micro-architectural behavior follows similar trends for TPC-C for the multi-threaded execution compared to the single-threaded execution. *DBMS D* largely suffers from Icache stalls, and *Shore-MT*, *DBMS N*, and *Silo* mainly suffer from Dcache stalls. *DBMS M*, once again, suffers less from the Icache stalls than it does for the single-threaded execution. This is, once again, likely due to the instruction locality of the modules that are more active during the multi-threaded execution.

Table 7 Consumed bandwidth in GB/s as we increase the database size for multi-threaded execution when reading 1 row per transaction

	1 MB	10 MB	10 GB	100 GB
DBMS D	0	0	0	0
Shore-MT	0	0	2	2
DBMS M	0	0	1	-
DBMS N	5.2	5.1	6.2	6.2
Silo	0	0	8.2	8.3

8 Memory bandwidth consumption

This section presents the consumed memory bandwidth for the sensitivity to data size and to work per transaction micro-benchmarks and TPC-C benchmark. We measured both single- and multi-threaded consumed memory bandwidth. We observed that the consumed single-threaded bandwidth is always less than 1 GB/s for all the systems. Hence, we omit the single-threaded bandwidth results, and focus on the multi-threaded ones.

8.1 Data size micro-benchmark

Table 7 presents the consumed bandwidth for increasing data size. All the systems consume significantly lower memory bandwidth than the maximum available bandwidth. While the maximum available bandwidth is 66 GB/s, the maximum consumed bandwidth is 8.3 GB/s by *Silo* for 100 GB of data size.

DBMS D, *Shore-MT*, and *DBMS M* consume significantly lower bandwidth than *DBMS N* and *Silo*. *DBMS D* and *M* suffers from Icache stalls, which prevents them from stressing the memory bandwidth. *Shore-MT* suffers from Dcache stalls for 10 GB and 100 GB. As being a disk-based system, it nevertheless has a significantly larger instruction footprint and is significantly slower (see Table 2) than *DBMS N* and *Silo*. As a result, it stresses the memory bandwidth only modestly.

Table 8 Consumed bandwidth in GB/s as we increase the amount of work per transaction for multi-threaded execution for a database of size 100 GB (10 GB for DBMS N)

	1 row	10 rows	100 rows
DBMS D	0	2	3
Shore-MT	2	2	2.5
DBMS M	1	7	11
DBMS N	6.2	8.5	8.4
Silo	8.3	8.3	8.4

DBMS N has relatively high bandwidth consumption for 1 and 10 MB of data. This is due to *DBMS N*'s transaction setup processing. As the data size is increased the consumed bandwidth is also increased. As *DBMS N* spends more time on index lookup when the data size is increased, it consumes more memory bandwidth.

Silo consumes no memory bandwidth for 1 and 10 MB of data as the data is mostly cache-resident. *Silo* consumes the highest bandwidth among the systems we analyze for 10 and 100 GB of data. *Silo* eliminates the overheads of disk-based systems and also does not suffer from the cost of transaction setup and instantiation. As a result, it delivers the highest relative throughput (see Table 2) and stresses the memory bandwidth the highest. Nevertheless, *Silo*'s maximum consumed bandwidth is significantly less than the maximum available bandwidth of 66 GB/s. This shows that OLTP systems generate only modest amount of memory traffic and hence severely under-utilize the memory bandwidth.

8.2 Work per transaction micro-benchmark

Table 8 shows the consumed bandwidth as we increase the amount of work per transaction. We use 100 GB of database for all the systems, except *DBMS M* and 10 GB of database for *DBMS M*. The consumed bandwidth is increased as the amount of work per transaction is increased for all the systems. This increase is more pronounced for *DBMS M*. As the amount of work per transaction is increased *DBMS M* suffers less and less from the legacy code modules that it borrows from *DBMS D*. Hence, it suffers less and less from the Icache stalls. As a result, it's relative throughput and consumed bandwidth is significantly increased.

The increase is also observable for *DBMS D* and *DBMS N*. Both *DBMS D* and *DBMS N* suffer from the code outside the storage manager. As the number of rows read per transaction is increased, the effects of the code outside the storage manager is reduced. Hence, the stress on the memory bandwidth is increased.

The increase in the consumed bandwidth is significantly less for *DBMS D* than it is for *DBMS M*. *DBMS D*'s legacy codebase overheads are heavier than they are for *DBMS M*. As a result, the increased amount of work inside the transaction mitigates the overheads of the code outside the storage manager only partially. This effect is more clean in Fig. 6. As the figure shows, *DBMS D* spends ∼35% of its time outside the storage manager when reading 100 rows per transaction.

Shore-MT and *Silo*'s consumed bandwidths are only modestly increased. This is because *Shore-MT* and *Silo* hard code transactions in C++. Hence, the increased amount of work per transaction stresses the memory bandwidth at a similar level per unit of a time. As a result, the consumed bandwidth remains mostly stable as the amount of work per transaction is increased.

Overall, despite the increased amount of work per transaction, all the OLTP systems we examine consume only a modest fraction of the maximum available bandwidth. While the maximum available bandwidth is 66 GB/s, the maximum consumed bandwidth is 11 GB/s by *DBMS M* when reading 100 rows per transaction.

Table 9 Consumed bandwidth in GB/s for TPC-C benchmark for multi-threaded execution

	TPC-C
DBMS D	0
Shore-MT	2.6
DBMS M	0
DBMS N	2.6
Silo	5.3

8.3 TPC-C

In this section, we examine the amount of consumed bandwidth for TPC-C benchmark for a database of size 100 GB, except for *DBMS M*. We use 32 GB of database for *DBMS M*. Table 9 shows the results. The consumed bandwidth values are less for all the systems compared to the micro-benchmark. This is expected as TPC-C transactions are more complex with more workload locality, and hence require more on-chip computation rather than stressing memory.

DBMS D and *M*, being Icache-stalls-bound systems, consume very low memory bandwidth. *Shore-MT*, *DBMS N*, and *Silo*, being Dcache- and resource/dependency-stalls-bound, consume certain amount of memory bandwidth. *Silo*, being the fastest OLTP system we analyze, consumes the highest amount of memory bandwidth. Unlike the micro-benchmark, *DBMS N* consumes a similar amount of bandwidth to *Shore-MT*. This is due to *DBMS N*'s increased time spent on tuple serializing/deserializing, preventing *DBMS N* from creating higher memory traffic. Nevertheless, all the systems we analyze consume bandwidth that is significantly below the maximum available bandwidth. While maximum available bandwidth is 66 GB/s, the highest consumed bandwidth is 5.3 GB/s by *Silo*.

Table 10 Normalized throughput values for hyper-threading evaluation

	Micro-bench.		TPC-C	
	ST	MT	ST	MT
DBMS M	1.2	1.3	1.2	1.3
Silo	1.4	1.6	1.4	1.7

9 Acceleration features

In this section, we examine three popular acceleration features that today's processors provide: hyper-threading, turbo-boost, and hardware prefetchers. We present normalized throughput values.

We use *DBMS M* and *Silo* when running the read-only micro-benchmark while probing 1 row per transaction, and when running the TPC-C benchmark for single- and multi-threaded executions. We use 10 GB of database for the micro-benchmark and 32 GB of database for TPC-C when profiling *DBMS M*. We use 100 GB of database for *Silo*.

9.1 Hyper-threading

Table 10 shows normalized throughput values for hyper-threading evaluation. For single-threaded execution, the normalized throughput shows the throughput improvement when running two threads on the same physical core compared to running a single thread on a single physical core. For multi-threaded execution, the normalized throughput shows the throughput improvement when running 28 threads on 14 physical cores compared to running 14 threads on 14 physical cores (assuming that 14 threads deliver the highest throughput). We use the DBMS's and/or OS's relevant configuration interface to bind the threads to a single socket and allocate memory locally.

We observe that hyper-threading is modestly useful for *DBMS M*, whereas it is significantly useful for *Silo*. Hyper-threading is the most useful when there are long-latency data stalls that can easily be overlapped. As *Silo* highly suffers from Dcache stalls, hyper-threading provides a more significant speedup for *Silo*. We also observe that the improved throughput is higher for multi-threaded execution than it is for single-threaded both for *DBMS M* and *Silo*. This is likely due to the increased sharing of the data structures at the last-level cache when running concurrently on the multiple cores.

9.2 Turbo-boost

Table 11 shows normalized throughput values for turbo-boost evaluation. We present the throughput values with turbo-boost turned on normalized to the ones with turbo-boost turned off.

Table 11 Normalized throughput values for turbo-boost evaluation

	Micro-bench.		TPC-C	
	ST	MT	ST	MT
DBMS M	1.3	1.1	1.2	1.2
Silo	1.2	1.1	1.2	1.1

Table 12 Normalized throughput values for prefetcher evaluation

	Micro-bench.		TPC-C	
	ST	MT	ST	MT
DBMS M	1.0	1.0	1.0	1.0
Silo	1.0	1.0	1.0	1.0

We observe that both *DBMS M* and *Silo* modestly benefit from turbo-boost. Turbo-boost provides the highest speedups when the computation is arithmetic-operation-heavy rather than memory-access-bound as it is the case for OLTP. As a result, both systems only modestly benefit from turbo-boost feature.

9.3 Hardware prefetchers

Table 12 shows normalized throughput values for prefetcher evaluation. We present the throughput values with prefetchers disabled normalized to the ones with prefetchers enabled. There are four hardware prefetchers that today's server processors provide: L1 next line, L1 streamer, L2 next line, and L2 streamer prefetchers [18]. We disable them all and enable them all.

We observe that prefetchers have no visible effect on the OLTP system performance. OLTP workloads are random-data-access-bound and have low spatial locality. As a result, the streamer prefetchers might not be providing a visible performance gain when enabled. *DBMS M* mainly suffers from Icache stalls. Hence, the improvement that the next line prefetchers bring is likely to be negligible. *Silo* uses software prefetching to prefetch consecutive cache lines that belong to the same index node, during its index traversal. Hence, the disabled next line prefetcher is likely not creating an observable effect on *Silo*'s throughput.

10 Ivy Bridge versus Broadwell

In this section, we compare two Intel generations in terms of their micro-architectural behavior when running the micro-benchmark that randomly reads 1 row and the TPC-C benchmark. We use 100 GB of database. We use 10 GB of a database for the micro-benchmark and 32 GB of a database for TPC-C for *DBMS M*. We examine the Intel Xeon v2 line Ivy Bridge micro-architecture and Intel Xeon v4 line Broad-

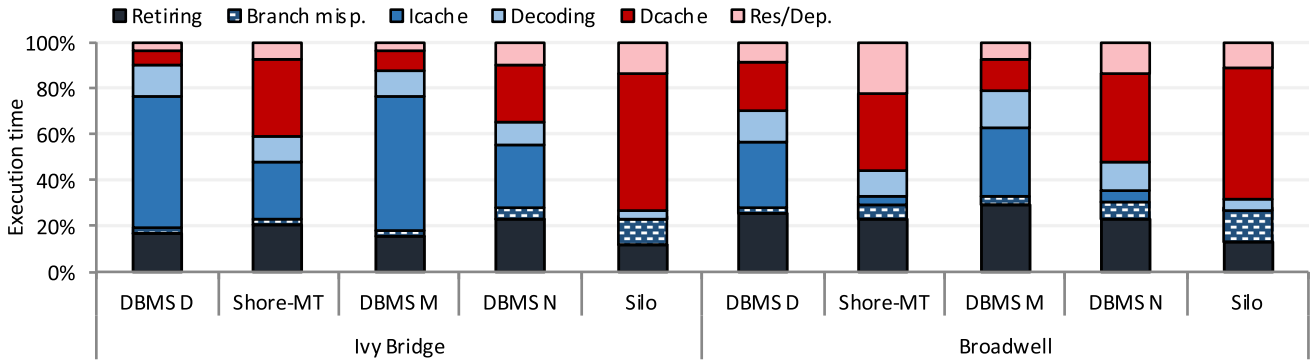


Fig. 15 Breakdowns of the CPU cycles for successive Intel micro-architectures when running micro-benchmark

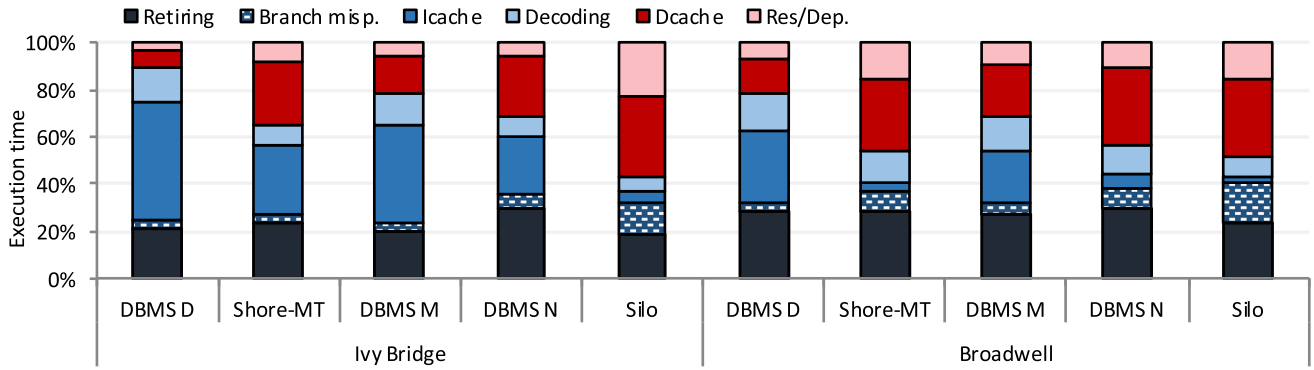


Fig. 16 Breakdowns of the CPU cycles for successive Intel micro-architectures when running TPC-C

well micro-architecture. We choose these two generations as there is major micro-architectural change from the Ivy Bridge to the Haswell micro-architecture, especially in the instruction fetch unit of the processors [14] (see Sect. 4.1.1, paragraph 2). Broadwell is slightly improved version of the Haswell micro-architecture. As OLTP systems are known to severely suffer from Icache stalls, we examine how effective the instruction fetch unit improvement for OLTP systems.

Figures 15 and 16 show the results. We observe that there is a significant change in the micro-architectural behavior of the OLTP systems across the two processor generations. On the Ivy Bridge micro-architecture, all the systems except for *Silo* mainly suffer from Icache stalls. On the Broadwell micro-architecture, *Shore-MT* and *DBMS N*'s main micro-architectural bottlenecks shift from Icache stalls to Dcache stalls. Similarly, *DBMS D* and *M* suffer significantly less from the Icache stalls on the Broadwell micro-architecture compared to the Ivy Bridge micro-architecture. This shows that the advances in the instruction fetch unit of the processor significantly help for reducing the Icache stalls.

Our finding on Ivy Bridge vs. Broadwell corroborates the recent work of Yasin et al. [60] where SPEC benchmarks are evaluated across Ivy Bridge and Skylake (the generation after Broadwell) micro-architectures. Yasin et al. also show that the improvement on the Skylake micro-architecture, which inherits the improvements from the Broadwell micro-architecture, significantly reduces the Icache stalls.

Table 13 Throughput on Broadwell normalized to throughput on Ivy Bridge

	Micro-bench.	TPC-C
DBMS D	2.0	1.6
Shore-MT	1.6	1.6
DBMS M	2.5	1.4
DBMS N	1.6	1.5
Silo	1.1	1.3

Table 13 shows the throughput of the Broadwell machine normalized to the throughput of the Ivy Bridge machine. As expected, the Broadwell machine delivers significantly higher throughput than the Ivy Bridge machine thanks to its micro-architectural improvements.

Existing work on micro-architectural analysis of OLTP systems has mostly used an Ivy Bridge or an earlier micro-architecture generation. As a result, their conclusions were mostly referring to the instruction overheads of disk-based and in-memory OLTP systems [48,49,53,54]. In this paper, we take the existing work one step ahead and provide

conclusions on one of the latest generations of Intel micro-architectures.

11 Lessons learned

This section summarizes the highlights of our work. In-memory OLTP systems implement a series of optimizations to reduce the instruction footprint and improve cache utilization. Despite all of the optimizations, they severely under-utilize the micro-architectural features similarly to the traditional disk-based systems.

Instruction stalls DBMS D and *M* incur the highest number of instruction stalls due to the large amount of legacy code they use. Hence, OLTP systems relying on legacy code modules should first optimize their instruction footprint. Transaction compilation is a promising technique that can be used to reduce the instruction footprint size and complexity of an OLTP system as shown by [34,48] (see Sect. 6.1).

Shore-MT, despite being a disk-based system, does not suffer from instruction cache miss stalls. Hence, today's processors are powerful enough to fetch and execute instruction stream of complex disk-based systems without stalling the instruction fetch unit. *DBMS N* and *Silo* do not suffer from the instruction stalls either. This shows that ground-up designed OLTP systems' instruction footprint is simple enough for today's processors to fetch instructions without stalling in the instruction cache (see Sect. 4.1.1).

Data stalls DBMS N and *Silo* mainly suffer from data cache stalls. *DBMS M* suffers from data stalls only when the instruction cache misses are mitigated by an increased instruction locality in their instruction stream (see Sect. 4.2). The data cache misses for *DBMS N* and *Silo* are mostly due to the random data accesses made during the index traversal. Hence, ground-up designed in-memory systems should firstly optimize their index structures. We have seen that Masstree [32] used by *Silo* significantly outperforms the red-black tree used by *DBMS N* (see Sect. 4.1.3, 4.2.1).

While using an efficient index structure improves the performance, its execution time is still dominated by data cache misses caused by the random data accesses during the index traversal. Hence, ground-up designed in-memory OLTP systems should adopt techniques that can mitigate the negative performance effect of random data accesses (see Sect. 4.2.1).

Mitigating data stalls One promising way to mitigate the random data accesses is using co-routines. Co-routines is a cheap thread interleaving mechanism that allows interleaving long-latency data stalls with computation. Psaropoulos et al. [38–40] and Jonathan et al. [19] have shown that co-routines can successfully be used to improve index join and index lookup.

Another promising technique to mitigate the random data accesses is using machine learning to learn the distribution of the keys and jump to the index location that the key belongs to without actually performing the index traversal. Kraska et al. [25], Sirin et al. [30], and Ding et al. [11] have shown that machine-learned indexes can successfully replace/accelerate the index search operation.

Row- versus column-oriented storage All the systems we profile use the row-oriented storage format. However, recently, several systems adopted the column-oriented storage, mostly to use a single format for both transactional and analytical processing [36]. Using column-oriented storage as opposed to row-oriented storage requires making multiple random data accesses per row access, as different attributes of the same row will be spread around the main-memory.

Today's processors are capable of overlapping random data accesses if their locations in the instruction stream are close to each other. Hence, making multiple random data accesses might not hurt the performance, if the data access primitives are well-implemented.

Alternatively, today's processors have the software prefetching capability. Programs can prefetch those memory blocks ahead of the access such that the memory access times are overlapped with useful work. Developers can use software prefetching to mitigate the negative effect of making multiple random data accesses.

On the other hand, bringing multiple cache lines to the processor cache might result in inefficiently using the processor caches. If an attribute is 8 bytes and we access four attributes of the same row, we bring $64 \times 4 = 256$ bytes (as each cache line is 64 bytes), instead of $8 \times 4 = 32$ bytes. The $256 - 32 = 224$ bytes of the data that is brought to the cache pollutes the cache, which can hurt the overall performance of the system.

Hardware In this study, we conclude that software-level optimizations do not directly translate into more efficient utilization of micro-architectural resources, and might even hinder it, on modern processors. One needs to optimize the hardware and software together as the next step putting micro-architectural utilization as a high priority goal.

Exhibiting low instruction- and memory-level parallelism, OLTP workloads are unable to utilize the wide-issue, complex out-of-order cores. Most of the time goes to the memory stalls for bringing either instructions or data items from the memory. Instruction cache sizes have been unchanged for the last decade due to the strict latency limitations, and we cannot expect them to increase. On the other hand, improved instruction fetch units can make a significant change in the micro-architectural behavior of the OLTP systems (see Sect. 10). Further advancements at the micro-architectural level, especially at the instruction fetch unit, can still have further potential impact to improve OLTP system performance, as popular OLTP systems such as *DBMS D* and *M* that we profiled still highly suffer from Icache stalls. Profile-guided (i.e., feedback-directed) optimization via the compiler and

hardware support for software code prefetching are shown to be effective for reducing Icache stalls for server workloads, which also significantly suffer from Icache stalls [4,8,59].

As ground-up designed OLTP systems mainly suffer from long-latency data cache stalls, hardware designers can invest more on hardware mechanisms that can overlap long-latency data stalls. Hyper-threading improves performance up to 70% in a carefully designed and implemented in-memory OLTP system (see Sect. 9.1). Turbo-boost and hardware prefetchers are modestly useful for OLTP workloads, as OLTP workloads are memory-latency-bound (see Sects. 9.2, 9.3). As the processor's power budget is limited, hardware designers can invest more power budget on the features that would overlap long-latency data stalls such as larger number of hyper-threads per physical core.

On the other hand, whatever the size of the last-level cache (LLC) is, megabytes of LLC will not be enough to keep the gigabytes of the data footprint of most standard OLTP benchmarks. Hence, instead of using beefy and complex out-of-order cores consuming large amount of energy, using simpler cores with intelligent hyper-threading mechanisms can improve the throughput of OLTP applications with a smaller power budget [12,13,31]. Sirin et al. [47] have shown that low-power ARM processor can provide 1.7 to 3 times lower throughput with 3 to 15 times less power consumption than a state-of-the-art Intel Xeon processor, achieving up to 9 times higher energy efficiency.

With that said, Kanev et al. [20] have shown that server workloads partially benefit from the wide-issue out-of-order execution. Hence, the use of wimpy cores with narrow-issue execution engines might produce a suboptimal performance and may not satisfy some application requirements. Similarly, Sirin et al. [47] have shown that ARM processors' quantified latency can be up to 11 times higher than Intel Xeon towards the tail of the latency distribution, which makes Intel Xeon more suitable for tail-latency-critical applications.

GPU/FPGA-based acceleration of database systems is another line of research that allows improving database performance by using alternative computing devices to power-hungry processors. [9,42,45] present techniques on using GPUs for accelerating analytical processing queries such as hash join. Kim et al. [24] have proposed a transaction processing engine architecture that exploits the wide-parallelism. Alonso et al. [3] present an open-source hardware-software co-design platform for database systems, which uses CPU and FPGA as the main building blocks. [21,44] present techniques on integrating FPGAs into common database operations such as data partitioning and regular expression. These studies highlight the opportunities to enrich the traditional computing space of database systems by alternative computing devices such as FPGAs and GPUs. As these computing devices provide massive parallelism and/or low-power consumption, they allow investigating the energy-efficiency space and potentially serve as the processors of the future database systems.

12 Conclusion

In this paper, we perform a detailed micro-architectural analysis of the in-memory OLTP systems contrasting them to the disk-based OLTP systems. Our study demonstrates that in-memory OLTP systems spend most of their time in stalls similarly to the disk-based OLTP systems despite all the design differences and lighter storage manager components of the memory-optimized systems. The lighter storage manager components reduce the instruction footprint at the storage manager layer, but the overall instruction footprint of an in-memory OLTP system is still large if the code base relies on legacy code modules. This leads to a poor instruction locality and high number of instruction cache misses. Ground-up designed in-memory OLTP systems can eliminate the instruction cache misses. In the absence of the instruction cache misses, the impact of long-latency data misses surfaces up, resulting in spending ∼70% of the execution time in stalls.

Acknowledgements We would like to thank the anonymous reviewers and the member of DIAS laboratory for their feedback. This project has received funding from the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa), and Swiss National Science Foundation, Project No.: 200021 146407/1 (Workload- and hardware-aware transaction processing).

Funding Open Access funding provided by EPFL Lausanne.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix A: Read-write micro-benchmark

In this appendix, we present the evaluation of the read-write version of the micro-benchmark.

A.1 Sensitivity to data size

In this section, we perform the sensitivity analysis for the database size. Figure 17 shows the CPU cycles breakdowns.

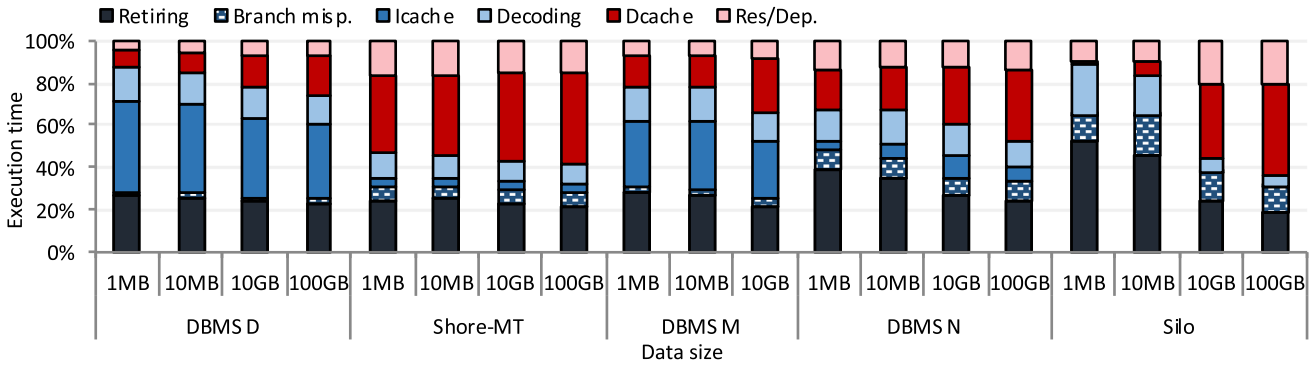


Fig. 17 Breakdowns of the CPU cycles as we increase the database size when running the read-write micro-benchmark

All the systems follow similar trends to what we observed for the read-only micro-benchmark, except *Shore-MT*. *Shore-MT* suffers significantly more from Dcache stalls for the read-write micro-benchmark than it is for the read-only micro-benchmark. We examined the function call trace of *Shore-MT*. The increased Dcache stalls are due to the logging costs that *Shore-MT* does for update queries, but not read-only queries. Being a disk-based system, *Shore-MT* uses a heavy data structure to keep a large amount of log information. *DBMS N*, on the other hand, uses command logging where only the invoked transaction and its parameters are logged. Hence, *DBMS N* suffers less from the logging operations.

DBMS M has higher Dcache stalls compared to the read-only micro-benchmark. This is likely due to that *DBMS M* uses a multi-version concurrency control mechanism, where updates are kept in deltas. Each new version of the data is written to a new memory location called delta, and the pointer to the record is updated to point to the location of the delta. Multiple versions are kept in multiple deltas that are chained one-after-the-other. Occasionally, the deltas are consolidated. Chain traversal/consolidation requires more random data accesses and hence more Dcache stalls.

DBMS N and *Silo* has less Dcache stalls compare to the read-only micro-benchmark. This is due to that update operation has a higher data locality than the read, as update requires reading from and writing to the same data block. As a result, it results in less Dcache stalls.

Table 14 shows the normalized throughputs for the read-write micro-benchmark. We normalize the values with respect to the read-only micro-benchmark, as the normalization across the systems provides similar results to Table 2. Read-write micro-benchmark is always slower. This is because the update operation requires more work for concurrency control and logging than the read-only operation. *Shore-MT*'s relative throughput is the lowest among the system. This highlights *Shore-MT*'s inefficient locking and logging mechanisms.

As the data size increases, read-write micro-benchmark's throughput gets closer to the read-only micro-benchmark.

Table 14 Normalized throughput for the read-write micro-benchmark. Throughput is normalized to the throughput values of the read-only micro-benchmark

	1 MB	10 MB	10 GB	100 GB
DBMS D	0.7	0.7	0.8	0.8
Shore-MT	0.5	0.5	0.6	0.6
DBMS M	0.7	0.7	0.7	–
DBMS N	0.7	0.7	0.7	0.8
Silo	0.8	0.8	0.8	0.9

This is because of the higher data locality of the read-write micro-benchmark. As the data size is increased, data locality matters more and more for the micro-benchmark's throughput. Nevertheless, read-only micro-benchmark's throughput is 20% to 50% higher than the read-write micro-benchmark.

A.2 Sensitivity to work per transaction

In this section, we perform the sensitivity analysis for the amount of work per transaction. Figure 18 shows the CPU cycles breakdowns. We observe similar trends to the read-only micro-benchmark As the number of rows updated per transaction is increased *DBMS D* and *M*'s Icache stalls are decreased, and they become more and more Dcache-stalls-bound. *Shore-MT*, *DBMS N* and *Silo*, being already Dcache-stalls-bound systems, have similar micro-architectural behavior across the varied amount of work per transaction.

Table 15 shows the normalized throughputs. We normalize the values with respect to the read-only micro-benchmark, as the normalization across the systems provides similar results to Table 3. We observe that read-write transaction throughput is decreased as the amount of work per transaction is increased for *DBMS D* and *M*. This is because the read-only micro-benchmark benefits more from the increased amount of work per transaction compared to the read-write micro-benchmark. The read-write micro-benchmark requires more

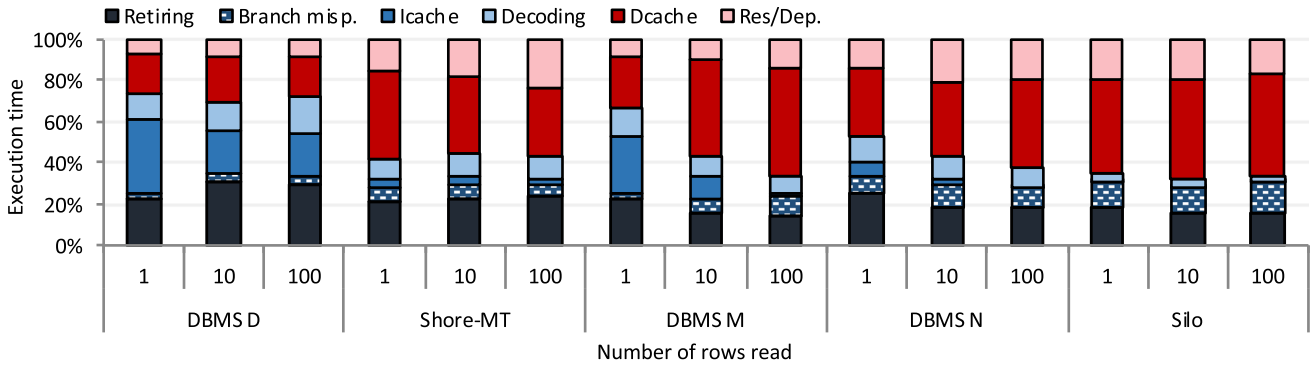


Fig. 18 Breakdowns of the CPU cycles as we increase the amount of work per transaction when running the read-write micro-benchmark

Table 15 Normalized throughput for the read-write micro-benchmark. Throughput is normalized to the throughput values of the read-only micro-benchmark

	1 row	10 rows	100 rows
DBMS D	0.8	0.8	0.7
Shore-MT	0.6	0.8	0.8
DBMS M	0.7	0.5	0.5
DBMS N	0.8	0.8	0.8
Silo	0.9	0.9	0.9

work and hence more instructions than the read-only micro-benchmark. As a result, the increased instruction locality is less useful to the read-write micro-benchmark than the read-only micro-benchmark.

Appendix B: Index, compilation, and data type

In this section, we evaluate the index, compilation and data type for the read-write micro-benchmark. Figure 19 presents CPU cycles breakdowns for different index types having compilation turned on and off. Compilation reduces the Icache stalls significantly similar to the read-only micro-benchmark. Table 16 presents the normalized throughput values. Compilation improves the throughput by 5–7.7× similar to the read-only micro-benchmark, showing how useful it is to reduce the instruction and the data footprint of *DBMS M*.

Figure 20 shows the CPU cycles breakdowns for long and string data types. Unlike the read-only micro-benchmark, DBMS M has similar amounts of Dcache stalls for long and string data types. The reason is the read-write micro-benchmark's increased Dcache stalls for the long data type. As a result, the overall CPU cycles breakdowns remains the same for the long and string data types. *DBMS N* and *Silo* present similar results to the read-only micro-benchmark.

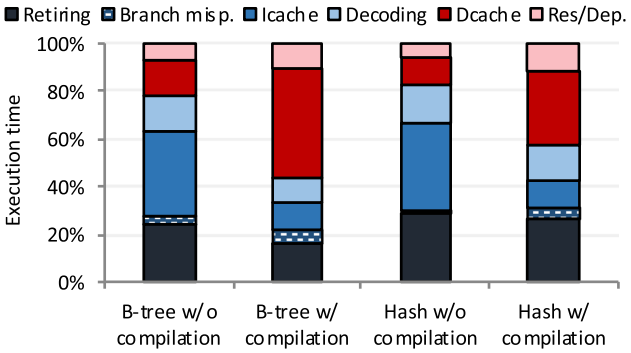


Fig. 19 Breakdowns of the CPU cycles for different index structures with and without compilation optimizations while running the read-write micro-benchmark

Table 16 Normalized throughput for different index structures with and without compilation when running the read-write micro-benchmark

	Micro-bench.
B-tree w/o comp.	1
B-tree w/ comp.	5
Hash w/o comp.	1.5
Hash w/ comp.	11.5

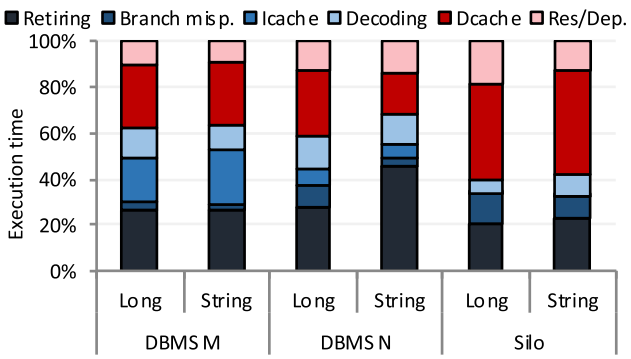


Fig. 20 Breakdowns of the CPU cycles for string and long data types while running the read-write micro-benchmark

Table 17 Normalized throughput for string and long data types while running the read-write micro-benchmark

	Long	String
DBMS M	1	0.6
DBMS N	1	0.9
Silo	1	0.6

DBMS N has less Dcache stalls for the string data type, whereas *Silo* has similar amount of Dcache stalls for the string and long data types.

Table 17 shows the normalized throughput values for string and long data types. The results are similar to the read-only micro-benchmark. While *DBMS M* and *Silo* have lower throughput for string due to the increased amount of work for string, *DBMS N* has similar throughput thanks to utilization of the workload locality for the string data type.

Appendix C: CPU cycles categorization

In this section, we present how we map each CPU cycles category that VTune provides to the individual categories that we use. Table 18 presents the mapping.

Table 18 The mapping between VTune's original and our simplified CPU cycles categorization

VTune's original category	Mapped category
Back-End, Memory	Dcache
Back-End, Core	Resource/dependency
Front-End, Front-End Latency, ICache Misses	Icache
Front-End, Front-End Latency, ITLB Overhead	Icache
Front-End, Front-End Latency, Branch Resteer	Branch misprediction
Front-End, Front-End Latency, DSB Switches	Decoding
Front-End, Front-End Latency, Length Changing Prefixes	Decoding
Front-End, Front-End Latency, MS Switches	Decoding
Front-End, Front-End Bandwidth	Decoding
Bad Speculation	Branch misprediction
Retiring	Retiring

References

1. TPC Transcation Processing Performance Council. http://www.tpc.org/
2. Ailamaki, A., DeWitt, D.J., Hill, M.D., Wood, D.A.: DBMSs on a Modern Processor: Where Does Time Go? In: VLDB, pp. 266–277 (1999)
3. Alonso, G., Roscoe, T., Cock, D., Ewaida, M., Kara, K., Korolija, D., Sidler, D., Wang, Z.: Tackling Hardware/Software co-design from a database perspective. In: CIDR (2020)
4. Ayers, G., Nagendra, N.P., August, D.I., Cho, H.K., Kanev, S., Kozyrakis, C., Krishnamurthy, T., Litz, H., Moseley, T., Ranganathan, P.: AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In: ISCA, pp. 462–473 (2019)
5. Barroso, L.A., Gharachorloo, K., Bugnion, E.: Memory system characterization of commercial workloads. In: ISCA, pp. 3–14 (1998)
6. Beamer, S., Asanovic, K., Patterson, D.: Locality Exists in graph processing: workload characterization on an Ivy bridge server. In: IISWC, pp. 56–65 (2015)
7. Bernstein, P.A., Goodman, N.: Multiversion concurrency control-theory and algorithms. ACM TODS **8**(4), 465–483 (1983)
8. Chen, D., Li, D.X., Moseley, T.: AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In: CGO, pp. 12–23 (2016)
9. Chrysogelos, P., Karpathiotakis, M., Appuswamy, R., Ailamaki, A.: HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. Proc. VLDB Endow. **12**(5), 544–556 (2019)
10. Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: SQL Server's Memory-optimized OLTP Engine. In: SIGMOD, pp. 1243–1254 (2013)
11. Ding, J., Minhas, U.F., Yu, J., Wang, C., Do, J., Li, Y., Zhang, H., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D., Kraska, T.: ALEX: An Updatable Adaptive Learned Index. Technical report (2020)
12. Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A.D., Ailamaki, A., Falsafi, B.: Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In: ASPLOS, pp. 37–48 (2012)
13. Haj-Yihia, J., Yasin, A., Asher, Y.B., Mendelson, A.: Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems. ACM Trans. Archit. Code Optim. **13**, 1 (2016)
14. Hammarlund, P., Martinez, A.J., Bajwa, A.A., Hill, D.L., Hallnor, E.G., Jiang, H., Dixon, M.G., Derr, M., Hunsaker, M., Kumar, R., Osborne, R.B., Rajwar, R., Singhal, R., D'Sa, R., Chappell, R., Kaushik, S., Chennupaty, S., Jourdan, S., Gunther, S., Piazza, T., Burton, T.: Haswell: the fourth-generation intel core processor. IEEE Micro **34**(2), 6–20 (2014)
15. Hardavellas, N., Pandis, I., Johnson, R., Mancheril, N., Ailamaki, A., Falsafi, B.: Database Servers on Chip Multiprocessors: Limitations and Opportunities. In: CIDR, pp. 79–87 (2007)
16. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: OLTP through the looking glass, and what we found there. In: SIGMOD, pp. 981–992 (2008)
17. Intel: Intel VTune Amplifier XE Performance Profiler. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/
18. Intel: Intel(R) 64 and IA-32 Architectures Optimization Reference Manual (2016)
19. Jonathan, C., Minhas, U.F., Hunter, J., Levandoski, J.J., Nishanov, G.V.: Exploiting coroutines to attack the killer nanoseconds. Proc. VLDB Endow. **11**(11), 1702–1714 (2018)
20. Kanev, S., Darago, J.P., Hazelwood, K., Ranganathan, P., Moseley, T., Wei, G.Y., Brooks, D.: Profiling a warehouse-scale computer. In: ISCA, pp. 158–169 (2015)
21. Kara, K., Giceva, J., Alonso, G.: FPGA-based data partitioning. In: SIGMOD, pp. 433–445 (2017)
22. Keeton, K., Patterson, D.A., He, Y.Q., Raphael, R.C., Baker, W.E.: performance characterization of a quad pentium pro SMP using OLTP workloads. In: ISCA, pp. 15–26 (1998)

23. Kemper, A., Neumann, T., Finis, J., Funke, F., Leis, V., Mühe, H., Mühlbauer, T., Rödiger, W.: Processing in the hybrid OLTP & OLAP main-memory database system hyper. IEEE DEBull **36**(2), 41–47 (2013)
24. Kim, K., Johnson, R., Pandis, I.: BionicDB: Fast and power-efficient OLTP on FPGA. In: EDBT, pp. 301–312 (2019)
25. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: SIGMOD, pp. 489–504 (2018)
26. Larson, P., Zwilling, M., Farlee, K.: The Hekaton memory-optimized OLTP engine. IEEE DEBull **36**(2), 34–40 (2013)
27. Lee, J., Muehle, M., May, N., Faerber, F., Sikka, V., Plattner, H., Krüger, J., Grund, M.: High-performance transaction processing in SAP HANA. IEEE DEBull **36**(2), 28–33 (2013)
28. Levandoski, J., Lomet, D., Sengupta, S.: The Bw-Tree: A B-tree for new hardware platforms. In: ICDE, pp. 302–313 (2013)
29. Lindstrom, J., Raatikka, V., Ruuth, J., Soini, P., Vakkila, K.: IBM solidDB: In-memory database optimized for extreme speed and availability. IEEE DEBull **36**(2), 14–20 (2013)
30. Llaveshi, A., Sirin, U., Ailamaki, A., West, R.: Accelerating B+tree search by using simple machine learning techniques. In: AIDB, pp. 1–10 (2019)
31. Lotfi-Kamran, P., Grot, B., Ferdman, M., Volos, S., Kocberber, O., Picorel, J., Adileh, A., Jevdjic, D., Idgunji, S., Ozer, E., Falsafi, B.: scale-out processors. In: ISCA, p. 500–511 (2012)
32. Mao, Y., Kohler, E., Morris, R.T.: Cache craftiness for fast multicore key-value storage. In: EuroSys, pp. 183–196 (2012)
33. MemSQL. http://www.memsql.com/
34. Neumann, T.: Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. VLDB Endow. **4**(9), 539–550 (2011)
35. Neumann, T., Leis, V.: Compiling database queries into machine code. IEEE DEBull **37**(1), 3–11 (2014)
36. Neumann, T., Mühlbauer, T., Kemper, A.: Fast serializable multi-version concurrency control for main-memory database systems. In: SIGMOD, p. 677–689 (2015)
37. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. PVLDB **3**(1), 928–939 (2010)
38. Psaropoulos, G., Legler, T., May, N., Ailamaki, A.: Interleaving with coroutines: a practical approach for robust index joins. Proc. VLDB Endow. **11**(2), 230–242 (2017)
39. Psaropoulos, G., Legler, T., May, N., Ailamaki, A.: Interleaving with Coroutines: thisatic and practical approach to hide memory latency in index joins. VLDB J. **28**(4), 451–471 (2019)
40. Psaropoulos, G., Oukid, I., Legler, T., May, N., Ailamaki, A.: Bridging the latency gap between NVM and DRAM for latency-bound operations. In: DaMoN, pp. 13:1–13:8 (2019)
41. Ranganathan, P., Gharachorloo, K., Adve, S.V., Barroso, L.A.: Performance of database workloads on shared-memory systems with out-of-order processors. In: ASPLOS, pp. 307–318 (1998)
42. Raza, A., Chrysogelos, P., Sioulas, P., Indjic, V., Anadiotis, A.G., Ailamaki, A.: GPU-accelerated data management under the test of time. In: CIDR (2020)
43. Shore-MT: Shore-MT Official Website. http://diaswww.epfl.ch/shore-mt/
44. Sidler, D., István, Z., Owaida, M., Kara, K., Alonso, G.: doppioDB: a hardware accelerated database. In: SIGMOD, pp. 1659–1662 (2017)
45. Sioulas, P., Chrysogelos, P., Karpathiotakis, M., Appuswamy, R., Ailamaki, A.: Hardware-conscious hash-joins on GPUs. In: ICDE, pp. 698–709 (2019)
46. Sirin, U., Ailamaki, A.: Micro-architectural analysis of OLAP: limitations and opportunities. Proc. VLDB Endow. **13**(6), 840–853 (2020)
47. Sirin, U., Appuswamy, R., Ailamaki, A.: OLTP on a server-grade ARM: power, throughput and latency comparison. In: DaMoN, pp. 10:1–10:7. ACM (2016)
48. Sirin, U., Tözün, P., Porobic, D., Ailamaki, A.: Micro-architectural analysis of in-memory OLTP. In: SIGMOD, pp. 387–402 (2016)
49. Sirin, U., Yasin, A., Ailamaki, A.: A Methodology for OLTP micro-architectural analysis. In: DaMoN, pp. 1:1–1:10 (2017)
50. Stets, R., Gharachorloo, K., Barroso, L.: A Detailed comparison of two transaction processing workloads. In: WWC, pp. 37–48 (2002)
51. Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., Helland, P.: The end of an architectural era: (It's Time for a Complete Rewrite). In: VLDB, pp. 1150–1160 (2007)
52. Stonebraker, M., Weisberg, A.: The VoltDB Main Memory DBMS. IEEE DEBull **36**(2), 21–27 (2013)
53. Tözün, P., Gold, B., Ailamaki, A.: OLTP in Wonderland—Where do cache misses come from in major OLTP components? In: DaMoN, pp. 8:1–8:6 (2013)
54. Tözün, P., Pandis, I., Kaynak, C., Jevdjic, D., Ailamaki, A.: From A to E: Analyzing TPC's OLTP Benchmarks—The Obsolete, The Ubiquitous, The Unexplored. In: EDBT, pp. 17–28 (2013)
55. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: SOSP, pp. 18–32 (2013)
56. VoltDB. http://www.voltdb.com
57. Wenisch, T.F., Ferdman, M., Ailamaki, A., Falsafi, B., Moshovos, A.: Temporal streams in commercial server applications. In: IISWC, pp. 99–108 (2008)
58. Yasin, A.: A top-down method for performance analysis and counters architecture. In: ISPASS, pp. 35–44 (2014)
59. Yasin, A., Ben-Asher, Y., Mendelson, A.: Deep-dive Analysis of the Data Analytics Workload in CloudSuite. In: IISWC, pp. 202–211 (2014)
60. Yasin, A., Haj-Yahya, J., Ben-Asher, Y., Mendelson, A.: A metric-guided method for discovering impactful features and architectural insights for skylake-based processors. TACO **16**(4), 46:1–46:25 (2020)
61. Yu, X., Bezerra, G., Pavlo, A., Devadas, S., Stonebraker, M.: Staring into the Abyss: an evaluation of concurrency control with one thousand cores. PVLDB **8**(3), 209–220 (2014)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.